Technical Report
TTIC-TR-2010-1
March 2010

Dynamic Well-Spaced Point Sets

Umut A. Acar
Max-Planck Institute for Software Systems
umut@mpi-sws.org

Andrew Cotter
Toyota Technological Institute at Chicago
cotter@ttic.edu

Benoît Hudson
Toyota Technological Institute at Chicago
bhudson@ttic.edu

Duru Türkoğlu
University of Chicago
duru@cs.uchicago.edu

ABSTRACT

In a *well-spaced point set* the Voronoi cells all have bounded aspect ratio, i.e., the distance from the Voronoi site to the farthest point in the Voronoi cell divided by the distance to the nearest neighbor in the set is bounded by a small constant. Well-spaced point sets satisfy some important geometric properties and yield quality Voronoi or simplicial meshes that can be important in scientific computations. In this paper, we consider the dynamic well-spaced point sets problem, which requires computing the 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 worst-case $O(\log \Delta)$ 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 the runtime of the dynamic update algorithm is optimal in the worst case. Our algorithm generates size-optimal outputs: the resulting output sets are never more than a constant factor larger than the minimum size necessary. A preliminary implementation indicates that the algorithm is indeed fast in practice. To the best of our knowledge, this is the first time- and size-optimal dynamic algorithm for well-spaced point sets.

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 [28]. 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 [7, 17]. The Voronoi diagram of a well-spaced point set is also immediately useful for the control volume method [19].

Given a d-dimensional hypercube $B \subset \mathbb{R}^d$, we define the well-spaced point set problem as constructing a well-spaced output $\mathsf{M} \subset B$ that is a superset of a given set of input points $\mathsf{N} \subset B$. We can construct the output by extending the input set with so called *Steiner* points, taking care to insert as few Steiner points as possible. We call the output and the 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., [5, 9, 24]), with several recent results obtaining fast runtimes [12, 14, 15, 27]. 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., [8, 23]) or they are asymptotically no faster than running a static algorithm from scratch [11, 18, 20].

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 space to the distance between the closest pair of points in the input. 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 number). Our algorithm consumes linear space in the size of the output and our update runtime is optimal in the worst-case.

To solve the dynamic problem, we first present an efficient algorithm for constructing size-optimal, well-spaced supersets (sections 4, 5, and 6). To enable dynamization, in addition to the output, the algorithm constructs 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 it produces similar computation graphs and outputs with similar inputs, e.g., that differ by one point. We make this property precise by describing a *distance* measure between the computation graphs and bounding this distance by $O(\log \Delta)$ when inputs differ by a single point (lemma 7.5). 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 8). For the lower bound, we show that there exist inputs and modifications that require $\Omega(\log \Delta)$ Steiner points to be inserted to the output (section 9).

The approach of designing a stable algorithm and then providing a dynamic update algorithm based on change propagation is inspired by recent advances on *self-adjusting computation* (e.g., [2, 16]). 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 algorithm is an adaptation of the change propagation algorithm for the problem of well-spaced point sets. Self-adjusting computation is found to be effective in kinetic motion simulation of three-dimensional convex hulls [3]. Although these initial findings are empiricial, 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, which has been used effectively for a relatively broad range of algorithms (e.g., [6, 10, 22, 25]).

The efficiency of our dynamic update algorithm directly depends on stability. We design a stable algorithm that maintains several invariants. First, we structure the computation into $\Theta(\log \Delta)$ levels—ranks and colors—such that the operations in each level depend only on the previous levels [27]. Second, we pick Steiner points by making local decisions only, using clipped Voronoi cells [15]. 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 executing from scratch the static algorithm

with the modified input (lemma 8.2). Consequently, the output remains both well-spaced and size-optimal with respect to the modified input (theorem 8.3).

To assess the effectiveness of the proposed dynamic algorithm, we present a prototype implementation and report the results of a preliminary experimental evaluation (section 10). Our experimental results confirm our theoretical bounds, showing linear speedups over re-computing from scratch. These results suggest that a well-optimized implementation can perform very well in practice.

2 Preliminaries

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 scale and shift the point set N such that $B = [0, 1]^d$ becomes a bounding box.

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 $\mathcal{M} \subset B$. The *nearest-neighbor distance of v in $\mathcal{M}$*, written $\mathrm{NN}_{\mathcal{M}}(v)$, is the distance from v to the nearest other vertex in $\mathcal{M}$. The *Voronoi cell* of v in $\mathcal{M}$, written $\mathrm{Vor}_{\mathcal{M}}(v)$, consists of points $x \in B$ such that for all $u \in \mathcal{M}$, $|vx| \leq |ux|$. Following Talmor [28], a vertex is *ρ-well-spaced* if the intersection of its Voronoi cell with B is contained in the ball of radius $\rho\,\mathrm{NN}_{\mathcal{M}}(v)$ centered at v; $\mathcal{M}$ is ρ-well-spaced if every vertex in $\mathcal{M}$ is ρ-well-spaced. The *β-clipped Voronoi cell* of v, written $\mathrm{Vor}^{\beta}_{\mathcal{M}}(v)$, is the intersection of $\mathrm{Vor}_{\mathcal{M}}(v)$ with the ball of radius $\beta\,\mathrm{NN}_{\mathcal{M}}(v)$ centered at v [15]. For any $\beta > \rho$, we define the *(ρ, β) picking region* of v, written $\mathrm{Vor}^{(\rho,\beta)}_{\mathcal{M}}(v)$, as $\mathrm{Vor}^{\beta}_{\mathcal{M}}(v) \setminus \mathrm{Vor}^{\rho}_{\mathcal{M}}(v)$, the region of the Voronoi cell bounded by concentric balls of radius $\rho\,\mathrm{NN}_{\mathcal{M}}(v)$ and $\beta\,\mathrm{NN}_{\mathcal{M}}(v)$. A vertex u is a *(β-clipped) Voronoi neighbor* of v if the (β-clipped) Voronoi cell of v contains a point equidistant from v and u. Figure 1 illustrates some of these definitions.

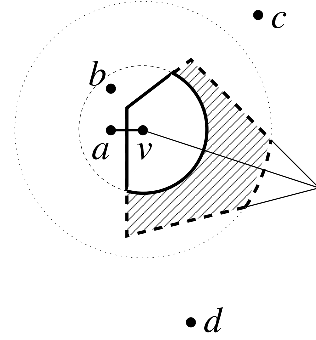

Figure 1: $\mathcal{M} = \{a, b, c, d, v\}$. $\mathrm{NN}_{\mathcal{M}}(v) = |va|$. Thick solid and dashed boundaries show $\mathrm{Vor}^{\rho}_{\mathcal{M}}(v)$ and $\mathrm{Vor}^{\beta}_{\mathcal{M}}(v)$, where $\rho = 2$ and $\beta = 4$. The ρ-clipped Voronoi neighbors of v are a and b. Shaded region is the (ρ, β) picking region of v.

Given an input set N, the *local feature size* of a point $x \in B$, written $\mathrm{lfs}(x)$, is the distance from x to the second-nearest vertex in N. The output M is *size-conforming* if there exists a constant c independent of N such that for all $v \in \mathsf{M}$, $\mathrm{NN}_{\mathsf{M}}(v) < c \cdot \mathrm{lfs}(v)$. Our algorithm guarantees that the output is size-conforming; this implies size-optimality [24].

3 Dynamic Balanced Quadtree

Our algorithm uses a point location structure based on the balanced quadtree of Bern, Eppstein, and Gilbert [5], which is relatively straightforward to dynamize and extend to d dimensions. We use "quadtree" to mean 2^d-tree and "quadtree node" to mean d-hypercube. Our well-spaced point set algorithm treats the quadtree almost as a black box; it uses only the leaves of the quadtree, which we refer to as *squares*. We define a *valid* d-dimensional balanced quadtree to be the *minimal* quadtree which satisfies the following three properties:

- Partitioning: every internal node contains all of its 2^d children.
- Crowding: every leaf node of the quadtree contains at most one vertex, and if it does, none of its neighbors contain a vertex.
- Grading: all neighbors of any internal node exist in the quadtree.

Here, we define the *neighbors* of a quadtree node to be the nodes in each of the $3^d - 1$ cardinal and diagonal directions, at the same level. In a valid quadtree, the set of squares partitions the space defined by the quadtree, and any two

adjacent squares, i.e, two squares that share a common border, are either neighbors or at consecutive levels. 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, and a list of Steiner vertices.

The quadtree supports the functions `QTBuild`, `QTAdd`, `QTRemove`, `QTApxNN`, and `QTClippedVoronoi`. The function $\texttt{QTBuild}(\mathsf{N})$ constructs a quadtree for a set of n vertices N in $O(n \log \Delta)$ time. It may be implemented most simply by calling `QTAdd` for each input vertex. The functions $\texttt{QTAdd}(\Pi, v^*)$ and $\texttt{QTRemove}(\Pi, v^*)$ respectively add or remove an input vertex v^* into or from N and update the quadtree Π to match the new input in $O(\log \Delta)$ time. They return the updated quadtree Π' and the set of squares of Π that are deleted or become internal quadtree nodes. The function $\texttt{QTApxNN}(v)$ returns the size (side length) $|s|$ of the quadtree square s containing v. The validity of the quadtree guarantees that this value is in $\Omega(\mathrm{NN}_{\mathsf{N}}(v))$ and less than $\mathrm{NN}_{\mathsf{N}}(v)$. The function $\texttt{QTClippedVoronoi}(v, \beta)$ returns the β-clipped Voronoi cell of v in $O(1)$ time under certain assumptions that our algorithm meets [15]. It also returns the nearest neighbor distance of v.

The `QTAdd` function may be implemented as follows: first, determine the square which contains the new vertex by performing a top-down traversal of the quadtree. If this square already contains an input vertex, then split it and descend into the child containing the new vertex, repeating as necessary. Finally, insert the new vertex into the resulting (currently empty) square. This entire operation requires $O(\log \Delta)$ time, as the depth of the quadtree is bounded by $O(\log \Delta)$. In order to restore validity, impose the crowding and grading conditions. For crowding, simply enumerate every square which could possibly be crowded by the new vertex, check if it is crowded, and if so, split it. The following lemma follows immediately from the definition of the crowding property; it implies that there are only a constant number of squares at each level, and hence $O(\log \Delta)$ overall, which need to be checked:

Lemma 3.1 *During a call to `QTAdd`, if a node must be split due to crowding, then either it or one of its neighbors contains the newly-inserted vertex.*

The grading condition may be imposed similarly: by enumerating all squares which could possibly be split due to grading in response to the insertion of the new vertex, checking the grading condition, and performing splits as necessary. The following lemma (very similar to lemma 1 of [21]), when combined with lemma 3.1, implies that there are only $O(\log \Delta)$ squares which need to be checked:

Lemma 3.2 *During a call to `QTAdd`, if a node must be split due to grading, then a descendent of one of its neighbors must have been split due to crowding.*

Proof: We prove this result by bounding the distance over which grading splits may "propagate" in the quadtree, in response to an originating crowding split. Suppose that ϕ' has been split due to crowding. Let ℓ' be the depth (in the quadtree) of ϕ', and δ_ℓ an upper bound on the distance in each coordinate from ϕ' to any node at depth $\ell < \ell'$ which must be split due to grading in response to ϕ' being split. We prove by induction on $\ell < \ell'$ that $\delta_\ell \leq 2^{-\ell}$. For the base case, $\ell = \ell' - 1$, the only nodes which must be split at depth ℓ are the nodes that are adjacent to ϕ', and they are at distance $\leq 2^{-\ell}$. By the definition of the grading property, if ϕ is a quadtree node at depth $\ell - 1$ which must be split, it follows that a neighbor of one of ϕ's children must have been split. The side length of ϕ's children is $2^{-\ell}$, and by the inductive assumption each of these children is at distance at most $\delta_\ell \leq 2^{-\ell}$ from ϕ'. Adding these two quantities yields that $\delta_{\ell-1} \leq 2^{-(\ell-1)}$, completing the induction. Because the side length of nodes at depth ℓ is $2^{-\ell}$, this result implies that if ϕ must be split at depth ℓ, then ϕ' is either a descendent of ϕ, or of one of its neighbors. ■

We have now shown that the entire `QTAdd` function may be implemented in $O(\log \Delta)$ time. We will prove one final result, which characterizes the squares which became internal nodes during the call to `QTAdd`:

Lemma 3.3 *For any square $s \in \Pi$ that is returned by $\texttt{QTAdd}(\Pi, v^*)$, we have $|sv^*| \in O(|s|)$.*

Proof: Lemmas 3.1 and 3.2 prove that every split square s is at most a neighbor's neighbor of the quadtree node containing v^* at the same level as s. Hence, $|sv^*| \leq 2\sqrt{d}|s|$. ■

The `QTRemove` function essentially performs the same steps as `QTAdd`, in reverse. It first descends through the quadtree in order to locate the quadtree square containing the vertex, and erases it. Next, motivated by lemmas 3.1 and 3.2, 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 3.3 holds for `QTRemove`:

Lemma 3.4 *For any square $s \in \Pi$ that is returned by* `QTRemove`(Π, v^*)*, we have $|sv^*| \in O(|s|)$.*

Proof: Every square returned by `QTRemove` is a child of a square which would be split by `QTAdd`, were the deleted vertex to be re-inserted into the quadtree. The proof of lemma 3.3 shows that all such squares satisfy the claim, so their children will also. ■

4 A Stable Algorithm

We can construct a well-spaced superset of an input set by repeatedly "filling" each vertex of the superset by applying a *fill* operation to it. When applied to some vertex v, which we say that it *acts on*, a fill operation makes the vertex ρ-well-spaced by inserting Steiner vertices into its Voronoi cell. Although correct, this basic algorithm is not efficient because the Voronoi cells can be arbitrarily complex (thus requiring super-constant time to compute), and because filling a vertex may adversely affect the well-spacedness of already processed vertices requiring them to be filled multiple times. This algorithm is also unstable, because inserting/deleting a single vertex into/from the input can result in very different outputs because the presence/absence of a vertex can affect the choice of many subsequent Steiner vertices.

To address these problems, we refine the basic algorithm to schedule carefully the fill operations such that 1) each fill operation requires constant time, 2) each vertex is filled at most once, 3) the algorithm is stable. To achieve these three properties, which we make precise and prove in the rest of the paper, we start by refining the fill operation so that instead of inserting points inside the Voronoi cell, it inserts points within the (ρ, β) picking region of the vertex that it acts on (figure 1). We then carefully order the fill operations so that no vertex is filled more than once and fill operations can be performed in constant time. These refinements yield an efficient algorithm. To ensure stability, we further refine the algorithm to identify certain fill operations as independent, which makes it possible to re-execute one operation without affecting another independent operation. In the rest of this section, we briefly describe these refinements and present the pseudo-code for the algorithm (figure 3).

Given a vertex set $\mathcal{M}$, consider applying a fill operation to a vertex $v \in \mathcal{M}$ that is not ρ-well-spaced. Let w be a Steiner vertex this operation inserts.

Fact 1 *The Steiner vertex w is in $\mathrm{Vor}_{\mathcal{M}}^{(\rho,\beta)}(v)$. That is, $\forall u \in \mathcal{M}$, $|wv| \leq |wu|$ and $\rho\,\mathrm{NN}_{\mathcal{M}}(v) \leq |wv| < \beta\,\mathrm{NN}_{\mathcal{M}}(v)$.*

Since v is the nearest neighbor of w, this fact implies that $\mathrm{NN}_{\mathcal{M}}(w) \geq \rho\,\mathrm{NN}_{\mathcal{M}}(v)$. Generalizing this simple observation, we infer the following.

Fact 2 *For any given $\alpha > 0$ if every vertex $u \in \mathcal{M}$ with $\mathrm{NN}_{\mathcal{M}}(u) < \alpha$ is ρ-well-spaced then $\mathrm{NN}_{\mathcal{M}}(w) \geq \rho\alpha$.*

Suppose that vertices whose nearest neighbors are at distance less than α are all ρ-well-spaced. The second fact implies that 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 $\rho\alpha$. Taking advantage of this property, we partially order the vertices by assigning *ranks* to them. More precisely, we define the rank of a vertex $v \in \mathcal{M}$ as the logarithm in base ρ of its nearest neighbor distance, i.e., $\lfloor \log_\rho \mathrm{NN}_{\mathcal{M}}(v) \rfloor$. We then fill the vertices in the order of their ranks. With this partial ordering, for example, the fill operations acting on 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 a logarithmic number of levels, $O(\log \Delta)$ in particular. As we prove in lemma 6.3, this ordering ensures that fill operations run in $O(1)$ time.

We ensure stability by identifying independent fill operations. We say that two fill operations at rank r are *independent* if, when executed (in either order), no operation inserts a Steiner vertex that becomes a β-clipped Voronoi neighbor of the vertex acted on by the other. We identify independent fill operations 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 fill operation at rank r that acts on a vertex v has color $c \in \{1, 2, \ldots, \kappa^d\}$ if v lies in a c colored r-tile. Figure 2 illustrates a coloring scheme. By choosing $\ell(r)$ small enough and κ large enough, we prove that two fill operations at the same rank are independent if they have the same color (lemma 7.1).

```
Dimension:  d, Parameters:  ρ,β,κ,ℓ(r)
StableWS (N) =
  Ω ← ∅ ; Π ← QTBuild(N)
  for each v ∈ N do
    Ω ← Ω ∪ {NewOp(v, ⌊log_ρ QTApxNN(v)⌋ , 0, nil)}
  for r = min rank in Ω to ⌊log_ρ √d⌋ do
    for each op ∈ Ω|_{r,0} do Dispatch (op,Ω)
    for c = 1 to κ^d do
      for each op ∈ Ω|_{r,c} do Fill (op,Ω)
  return (N,Π)

Fill (op,Ω) =
  (v, nnv, CV) ← QTClippedVoronoi(op,β)
  while v is not ρ-well-spaced (via CV) do
    choose w ∈ CV such that |vw| ≥ ρ · nnv
    op.steiners ← op.steiners ∪ {w}
    Ω ← Ω ∪ {NewOp(w, ⌊log_ρ |wv|⌋ , 0, op)}
    update CV with w

Dispatch (op,Ω) =
  (v, nnv, CV) ← QTClippedVoronoi(op,β)
  r ← ⌊log_ρ nnv⌋
  if r ≥ op.rank then
    Ω ← Ω ∪ {NewOp(v, r, Color(v, r), op)}
  for each β-clipped Voronoi neighbor w of v do
    r ← ⌊log_ρ |wv|⌋
    if r ≥ op.rank then
      Ω ← Ω ∪ {NewOp(w, r, Color(w, r), op)}

Color (v,r) =
  for i = 1 to d do c_i ← ⌊v_i/ℓ(r)⌋  mod κ
  return (c_1, c_2, ..., c_d) as a d digit number

NewOp (v, r, c, parent) =
  op ← CreateOp(v) ; op.rank ← r ; op.color ← c
  parent.children ← parent.children ∪ {op}
  op.children, op.steiners ← ∅ ; return op
```

Figure 3: Pseudo-code for our stable algorithm.

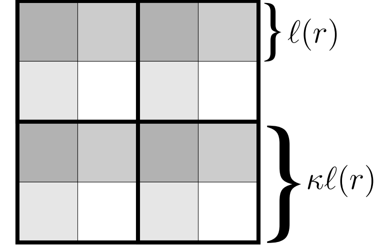

Figure 2: Illustration of a coloring scheme in 2D ($\kappa = 2$).

Figure 3 shows the pseudo-code of the algorithm. The pseudo-code follows the description quite closely except for the computation of ranks. Our algorithm critically relies on ordering the computation by assigning ranks to vertices and filling them in that order. 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 ranks dynamically. To achieve this, we assign ranks to fill operations and use another type of operation, called *dispatch*, to compute and update ranks. The unique dispatch operation acting on a vertex v also has a rank and runs before the fill operations acting on v. The rank of a dispatch operation acting on an input vertex v is an $O(1)$-approximation (from below) of the rank of v, and those that act on Steiner vertices are assigned exact ranks. When executed, a dispatch operation computes the rank of v, creates a fill operation for v at that rank, and creates fill operations for its β-clipped Voronoi neighbors in order to update their ranks. This approach guarantees that after the execution of the dispatch operations at rank r, every vertex either has a fill operation at its up-to-date rank, or a dispatch operation at rank greater than r. When executed, a fill operation makes well-spaced the vertex it acts on, subsequent fill operations terminate immediately without inserting Steiner vertices. We prefer this approach because it simplifies the analysis by making explicit the dependencies between operations. As we prove in theorem 5.4 this algorithm computes correctly and efficiently a ρ-well-spaced superset of its input.

The algorithm `StableWS` starts by constructing a quadtree Π and stores it for use in dynamic updates. It then computes the output M by creating (via `NewOp`) and performing dispatch and fill operations which it stores in Ω. The algorithm assigns a rank and a color, the pair of which we refer to as *time*, to each operation and executes them in time order. The dispatch operations are assigned the color zero. In the pseudo-code, we use $\Omega|_{r,c}$ to refer to the operations with time (r, c). In the analysis, we refer to time as a single entity rather than its components (rank and color). For brevity, we define time $t = 0$ to be the beginning of time, when the dispatch operations for the input are created but before any operations are performed, and 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, while executing, the algorithm constructs a computation graph of all executed operations and dependencies between them. The *computation graph* $G = (V, E)$ consists of nodes, $V = \Sigma \cup \Omega$, comprised of the set of squares (Σ) and the set of all operations (Ω), and directed edges representing various dependencies between operations and squares. Consider executing an operation op. If op creates an operation op' then (op, op') becomes an edge (recorded by storing op' in the `children` field of op). If op reads a square s via

`QTClippedVoronoi` then (s, op) becomes an edge (recorded in the square s). Finally, if op writes into a square s by inserting a Steiner vertex w into it then (op, s) becomes an edge (recorded by storing w in the `steiners` field of op). We tag each edge with the time of the operation that creates it, in the example above, this is the time of op.

5 Output Quality and Size

This section includes the proofs on the quality of the output of our algorithm, that M is ρ-well-spaced and size-optimal. Lemma 5.3 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 such an order that satisfies the key invariant that after filling a vertex, it becomes and remains ρ-well-spaced. Therefore, our algorithm incrementally progresses towards a ρ-well-spaced output. In these two lemmas, let $\mathcal{M}$ be the set of vertices in the output at the beginning of rank r.

Lemma 5.1 *At the beginning of rank r, assume that every vertex $u \in \mathcal{M}$ with $\mathrm{NN}_{\mathcal{M}}(u) < \rho^r$ is ρ-well-spaced. Then, for every vertex $w \in \mathcal{M}$ with $\mathrm{NN}_{\mathcal{M}}(w) \in [\rho^r, \rho^{r+1})$, there exists a fill operation that acts on w at rank r.*

Proof: Pick a vertex $w \in \mathcal{M}$, let u be its nearest neighbor in $\mathcal{M}$, and assume that $\rho^r \leq |wu| < \rho^{r+1}$. Let op_w and op_u be the dispatch operations that act on w and u respectively. If op_w runs at rank $\leq r$ and u is in the output when op_w is executed then op_w schedules a fill operation that acts on w at rank r. Alternatively, op_u schedules such a fill operation if op_u runs at rank $\leq r$ and w is a β-clipped Voronoi neighbor of u when op_u is executed.

Analyzing the vertices w and u, in two cases, we prove that the first condition holds. In the first case, if both w and u are input vertices then op_w runs at rank $\leq r$. In the second case, that w is a Steiner vertex and u is in the output when w is being created, consider the vertex v that creates w. By fact 1, we know that $|wv| \leq |wu|$, which implies that op_w runs at rank $\leq r$.

We prove that the second condition holds in the remaining case, that u is a Steiner vertex and that w is already in the output when u is being created. Similar to the previous case, we deduct that op_u runs at rank $\leq r$. Since u is the nearest neighbor of w in $\mathcal{M}$, w is a Voronoi neighbor of u in $\mathcal{M}'$, where $\mathcal{M}' \subset \mathcal{M}$ is the set of vertices in the output when op_u is executed. If u is ρ-well-spaced in $\mathcal{M}$ then $|wu| \leq 2\rho\,\mathrm{NN}_{\mathcal{M}}(u) < 2\beta\,\mathrm{NN}_{\mathcal{M}'}(u)$. Otherwise, the assumption of the lemma implies $\rho^r \leq \mathrm{NN}_{\mathcal{M}}(u)$. Since $|wu| < \rho^{r+1}$, we get $|wu| < \rho\,\mathrm{NN}_{\mathcal{M}}(u) < 2\beta\,\mathrm{NN}_{\mathcal{M}'}(u)$. Either way, w is a β-clipped Voronoi neighbor of u in $\mathcal{M}'$. ■

Lemma 5.2 (Progress) *At the beginning of rank r, every vertex $u \in \mathcal{M}$ with $\mathrm{NN}_{\mathcal{M}}(u) < \rho^r$ 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 $u \in \mathcal{M}$ with $\mathrm{NN}_{\mathcal{M}}(u) < \rho^r$ is ρ-well-spaced. For rank $r+1$, let $\mathcal{M}' \supset \mathcal{M}$ be the set of vertices in the output at the beginning of rank $r+1$ and consider a vertex $w \in \mathcal{M}'$ with $\mathrm{NN}_{\mathcal{M}'}(w) < \rho^{r+1}$. If $w \in \mathcal{M}' \setminus \mathcal{M}$ then w is a Steiner vertex inserted at rank r. Repeatedly applying fact 2 for each (Steiner) vertex in $\mathcal{M}' \setminus \mathcal{M}$, we see that the nearest neighbors of these Steiner vertices are at distance $\geq \rho^{r+1}$; in particular, $\mathrm{NN}_{\mathcal{M}'}(w) \geq \rho^{r+1}$. This is a contradiction, thus, $w \in \mathcal{M}$. Furthermore, $\mathrm{NN}_{\mathcal{M}}(w) < \rho^{r+1}$ for similar reasons. If $\mathrm{NN}_{\mathcal{M}}(w) < \rho^r$ then by our induction hypothesis w is ρ-well-spaced. Otherwise, by lemma 5.1, there exists a fill operation that acts on w at rank r. After executing that operation, w becomes ρ-well-spaced. Finally, fact 2 implies that w remains ρ-well-spaced. Therefore, our claim holds. ■

Lemma 5.3 *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 $v \in \mathsf{M}$, we have $c\,\mathrm{NN}_{\mathsf{M}}(v) \geq \mathrm{lfs}(v)$, 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 $v \in \mathcal{M}$, we have $c\,\mathrm{NN}_{\mathcal{M}}(v) \geq \mathrm{lfs}(v)$. Furthermore, assume that v inserts a Steiner vertex w and the new output is $\mathcal{M}' = \mathcal{M} \cup \{w\}$. We analyze the inductive claim for w and for any vertex $u \in \mathcal{M}$ separately. For w, by fact 1 we know that $|wv| \geq \rho\,\mathrm{NN}_{\mathcal{M}}(v)$ and $\mathrm{NN}_{\mathcal{M}'}(w) = |wv|$. By the triangle inequality, lfs satisfies the Lipschitz condition: $\mathrm{lfs}(v) + |wv| \geq \mathrm{lfs}(w)$. By the inductive hypothesis, $c\,\mathrm{NN}_{\mathcal{M}}(v) \geq \mathrm{lfs}(v)$. Therefore, we have $(\frac{c}{\rho} + 1)|wv| = (\frac{c}{\rho} + 1)\,\mathrm{NN}_{\mathcal{M}'}(w) \geq \mathrm{lfs}(w)$.

For any vertex $u \in \mathcal{M}$, if $\mathrm{NN}_{\mathcal{M}}(u) = \mathrm{NN}_{\mathcal{M}'}(u)$ then the claim holds trivially. Otherwise, assume that $\mathrm{NN}_{\mathcal{M}}(u) > \mathrm{NN}_{\mathcal{M}'}(u) = |wu|$. By the Lipschitz condition, we have $|wu| + \mathrm{lfs}(w) \geq \mathrm{lfs}(u)$ and by fact 1 we know $|wu| \geq |wv|$. Combining these by the bound we obtained for $\mathrm{lfs}(w)$, we get $(\frac{c}{\rho}+2)|wu| = (\frac{c}{\rho}+2)\,\mathrm{NN}_{\mathcal{M}'}(u) \geq \mathrm{lfs}(u)$. 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 [24]. ∎

Theorem 5.4 *`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 5.3 proves the size bound. ∎

6 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 6.1) proves that throughout the algorithm, the nearest neighbor distance of a vertex v changes only by a constant factor. The second (lemma 6.2) 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 or too late.

Lemma 6.1 *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 5.3), so $\mathrm{NN}_0(v) = \mathrm{lfs}(v) \in O(\mathrm{NN}_\infty(v))$. For a Steiner vertex w that is created at time $t = (r, c)$, fact 1 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 6.2 *If an operation at rank r acts on v then* $\mathrm{NN}_\infty(v) \in \Theta(\rho^r)$.

Proof: Consider an operation *op* that acts on a vertex v at time $t = (r, c)$. If *op* is a dispatch operation and v is an input vertex then the call `QTApxNN`(v) returns a value in $\Theta(\mathrm{NN}_0(v))$ which implies $\mathrm{NN}_0(v) \in \Theta(\rho^r)$. By lemma 6.1, we know that $\mathrm{NN}_0(v) \in \Theta(\mathrm{NN}_\infty(v))$, therefore, the result follows.

Otherwise, let *op*$'$ be the operation that creates *op*, and assume that *op*$'$ acts on u at time $t' = (r', c')$. Since *op*$'$ schedules *op* to be executed at rank r, we know that $r = \lfloor \log_\rho |uv| \rfloor$. Since $\mathrm{NN}_t(v) \leq |uv|$, we get $\mathrm{NN}_t(v) < \rho^{r+1}$. Thus, the upper bound holds: $\mathrm{NN}_\infty(v) \leq \mathrm{NN}_t(v) \in O(\rho^r)$. For the lower bound, from lemma 5.3, we have $\mathrm{NN}_\infty(v) \in \Omega(\mathrm{lfs}(v))$. By definition, $\mathrm{lfs}(v) \geq \mathrm{NN}_{t'}(v)$, and since $r = \lfloor \log_\rho |uv| \rfloor$, we have $|uv| \geq \rho^r$. Thus, it suffices to show that $\mathrm{NN}_{t'}(v) \in \Omega(|uv|)$. Since *op*$'$ creates *op*, we know that v is a β-clipped Voronoi neighbor of u at time t', which means that u is a Voronoi neighbor of v at time t'. If $\mathrm{NN}_{t'}(v) < \rho^{r'}$ then by Progress Lemma, v is ρ-well-spaced at time t'. Therefore, $2\rho\,\mathrm{NN}_{t'}(v) \geq |uv|$ and we are done. If $\mathrm{NN}_{t'}(v) \geq \rho^{r'}$, we know that $|uv| \leq 2\beta\,\mathrm{NN}_{t'}(u)$ because v is a β-clipped Voronoi neighbor of u. Applying the upper bound result from above for *op*$'$, we get $\mathrm{NN}_{t'}(u) \in O(\rho^{r'})$, thus, $|uv| \in O(\rho^{r'})$. Since $\mathrm{NN}_{t'}(v) \geq \rho^{r'}$, this implies $\mathrm{NN}_{t'}(v) \in \Omega(|uv|)$. ∎

Lemma 6.3 *Every operation runs in $O(1)$ time.*

Proof: Pick an operation acting on v at time $t = (r, c)$. The main costs are the `QTClippedVoronoi` calls and the loops. The Progress Lemma shows that every vertex $u \in \mathsf{M}_t$ with $\mathrm{NN}_t(u) < \rho^r$ is ρ-well-spaced and lemmas 6.2 and 6.1 together show that $\mathrm{NN}_t(v) \in \Theta(\rho^r)$. Hudson and Türkoğlu show that these are sufficient conditions to guarantee that `QTClippedVoronoi` runs in constant time [15].

The dispatch operation iterates over each β-clipped Voronoi neighbors. Since `QTClippedVoronoi` runs in constant time, there is only $O(1)$ neighbors. The fill operation has a loop that inserts Steiner vertices until v is

ρ-well-spaced. For each inserted Steiner vertex w, fact 1 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 each fill operation inserts $O(1)$ Steiner vertices. ■

Lemma 6.4 *For every vertex $v \in \mathsf{M}$, there are $O(1)$ operations that act on v.*

Proof: By lemma 6.2, we know that any operation that acts on v has rank $\lfloor \log_\rho \mathrm{NN}_\infty(v) \rfloor \pm O(1)$. Therefore, if we can show that the number of the operations that acts on v at each rank is constant, our claim will hold. There is only one dispatch operation for each vertex, so we only need to count fill operations scheduled by other dispatch operations. Fix r and consider a dispatch operation at time $t' = (r', 0)$ that acts on u and schedules a fill operation that acts 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 6.1 and 6.2 show that $\mathrm{NN}_{t'}(u) = O(\rho^{r'})$. These facts altogether imply $\rho^r = O(\rho^{r'})$. Again by lemma 6.2, 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 6.5 `StableWS` *runs in $O(n \log \Delta)$ time.*

Proof: As proven in section 3, building the quadtree takes $O(n \log \Delta)$ time. By lemmas 6.3 and 6.4, 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. ■

7 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 7.5).

As described in section 4, `StableWS(N)` creates 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 `StableWS(N')` and creating $G' = (V', E')$, Σ', and Ω' similarly. We define two squares $s \in \Sigma$ and $s' \in \Sigma'$ to be *identical*, written $s \equiv s'$, if s and s' have the same corner points. Also, two operations $op \in \Omega$ and $op' \in \Omega'$ are *identical*, written $op \equiv op'$, if op and op' have the same time and act on the same vertex. There exists a unique function $\mu : V' \to V$, $\mu = \mu_o \cup \mu_s$, where μ_o is the largest set satisfying $\mu_o = \{(op', op) \mid op' \in \Omega' \wedge op \in \Omega \wedge (op' \equiv op) \wedge (\mathrm{parent}(op'), \mathrm{parent}(op)) \in \mu_o\}$ and $\mu_s = \{(s', s) \mid s' \in \Sigma' \wedge s \in \Sigma \wedge s' \equiv s\}$. We call μ the *matching* between G' and G. Informally, μ pairs squares of G' with identical squares of G and pairs operations of G' with identical operations of G as long as their parents (the operations that create them, if any) are also paired. We say that nodes $u' \in V'$ and $u \in V$ *match* if $\mu(u') = u$. We denote the domain and the range of μ by $\mathsf{dom}(\mu)$ and $\mathsf{range}(\mu)$.

Given $G = (V, E)$ and $G' = (V', E')$ and their matching μ, let $\mu' = \mu \cup \{(u, u) \mid u \in V' \setminus \mathsf{dom}(\mu)\}$ 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 $\mu'(E') = \{(\mu'(u), \mu'(v)) \mid (u, v) \in E'\}$. 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 7.1 allows us to refine this definition: a path $(u_0, u_1, \ldots, u_h)$ is a *dependency path* if the times of the edges $(u_0, u_1), (u_1, u_2), \ldots, (u_{h-1}, u_h)$ increase monotonically.

Lemma 7.1 *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 they have the same color.*

Proof: Consider two fill operations, op_v and op_u, at the same rank and color acting on vertices v and u respectively. Let r be the rank of these operations and $\mathcal{M}$ be the set of vertices in the output at the beginning of rank r. If both v and u are ρ-well-spaced in $\mathcal{M}$ then op_v and op_u do not insert any Steiner vertices. Thus, op_v and op_u are independent. Otherwise, if v is not ρ-well-spaced the Progress Lemma implies that $\mathrm{NN}_{\mathcal{M}}(v) \geq \rho^r$. 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 op_v and op_u have the same color, v and u are far apart, more precisely, $|vu| \geq (\kappa - 1)\ell(r) > 3\beta\rho^{r+1}$. By fact 1, we know that any Steiner vertex w that op_v inserts satisfies $|vw| \leq \beta\,\mathrm{NN}_{\mathcal{M}}(v)$. By the existence of op_v and op_u, we already know $\mathrm{NN}_{\mathcal{M}}(v), \mathrm{NN}_{\mathcal{M}}(u) < \rho^{r+1}$. Using the triangle inequality, we get $|uw| \geq |vu| - |vw| > 2\beta\rho^{r+1} > 2\beta\,\mathrm{NN}_{\mathcal{M}}(u)$. The last inequality asserts that w cannot be a β-clipped Voronoi neighbor of u. Similar arguments can be made for u as well; therefore, the operations op_v and op_u 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 v^* of the symmetric difference of N and N'. We define an operation $op \in \mathsf{range}(\mu)$ 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 7.2 *For every operation in $\Omega^- \cup \Omega^+ \cup \Omega^{\times}$, there exists a dependency path from a square in $\Sigma^{\times}$.*

Proof: By definition of inconsistent operations, an operation $op \in \Omega^{\times}$ 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 op be the earliest of such operations. Let us assume that op is a dispatch operation acting on an input vertex v. Since op does not depend on an inconsistent square, it does not read an inconsistent square. Therefore, v is in $\mathsf{N} \cap \mathsf{N}'$ and lies in identical squares in both executions, which implies that `QTApxNN`(v) returns the same value for v in both executions and that their ranks are the same. Then, the definition of μ_o matches op with op' because op and op' are identical. Therefore, op is not a dispatch operation acting on an input vertex. Then, consider the operation op'' that creates op. By minimality of op, op'' can be reached via a dependency path from a square in $\Sigma^{\times}$. Extending that path to op proves the contradiction. ■

As proven by Hudson and Türkoğlu [15], the function `QTClippedVoronoi` satisfies the following locality property: for a given input N, a size-conforming set of vertices $\mathcal{M} \supset \mathsf{N}$, and a square s read by `QTClippedVoronoi`, for all $x \in s$, $|vx| \in O(\mathrm{NN}_{\mathcal{M}}(v))$. This property allows us to relate the operations on a dependency path geometrically.

Lemma 7.3 *Consider two operations op and op' in $G^{\cup}$ acting on vertices v and w. If there exists a dependency path from op' to op and op is at rank 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 the vertices on which they act. Consider an edge $e \in E$ with time $t_e = (r_e, c_e)$. The edge e consists of an operation $op_1 \in \Omega$ acting on v at time t_e and either a square s that it accesses (reads/writes) or another operation op_2 that it schedules. Using the locality result stated above, we bound the distance between op_1 and s by $O(\mathrm{NN}_{t_e}(v))$. Also, op_2 is within the same distance. Lemmas 6.1 and 6.2 bound $\mathrm{NN}_{t_e}(v)$ 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. The same analysis applies for any edge $e' \in E^{\cup}$.

By definition of dependency paths, the times of the edges on a dependency path from op' to op monotonically increase. Assuming that the rank of op' 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,

```
Global queues:  Ω⊖, Ω⊕, Ω⊗
PropagateWS (N, Σ⊗) =
  MarkReaders (Σ⊗, 0)
  for each s ∈ Σ⊗ and each v ∈ N ∩ vertices of s do
    Ω⊖ ← Ω⊖ ∪ {Dispatch of v}
    Ω⊕ ← Ω⊕ ∪ {NewOp(v, ⌊log_ρ QTApxNN(v)⌋, 0, nil)}
  for r = min rank in Ω⊖ ∪ Ω⊕ ∪ Ω⊗ to ⌊log_ρ √d⌋ do
    UndoOps (r, 0)
    for each op ∈ (Ω⊕ ∪ Ω⊗)|_{r,0} do Dispatch (op, Ω⊕)
    for c = 1 to κ^d do
      UndoOps (r, c)
      for each op ∈ (Ω⊕ ∪ Ω⊗)|_{r,c} do
        Fill (op, Ω⊕)
        S ← squares containing op.steiners
        MarkReaders (S, (r, c))

MarkReaders (Σ̂, t) =
  for each s ∈ Σ̂ and each op that reads s do
    if (op.rank, op.color) > t then Ω⊗ ← Ω⊗ ∪ {op}

Add (N, Π, v*) =
  (Π′, Σ⁻) ← QTAdd(Π, v*)
  op_v* ← NewOp(v*, ⌊log_ρ QTApxNN(v*)⌋, 0, nil)
  Ω⊕ ← {op_v*};  Ω⊖, Ω⊗ ← ∅
  PropagateWS(N, Σ⁻ ∪ {square of v*})
  return (N ∪ {v*}, Π′)

Remove (N, Π, v*) =
  (Π′, Σ⁻) ← QTRemove(Π, v*)
  Ω⊖ ← {Dispatch of v*};  Ω⊕, Ω⊗ ← ∅
  PropagateWS(N, Σ⁻ ∪ {square of v*})
  return (N \ {v*}, Π′)

UndoOps (r, c) =
  for each op ∈ (Ω⊖ ∪ Ω⊗)|_{r,c} do
    Ω⊖ ← Ω⊖ ∪ op.children
    S ← squares containing op.steiners
    MarkReaders (S, (r, c))
    remove all vertices in op.steiners
  Ω⊗ ← Ω⊗ \ Ω⊖|_{r,c}
  ResetEdges(Ω⊗|_{r,c})
```

Figure 4: Pseudo-code for the dynamic algorithm.

or an inconsistent operation that acts on it. Since there is a constant number of operations acting on a given vertex (lemma 6.4), the number of affected vertices measures the distance asymptotically. We define the sets of affected vertices in both executions: $\widehat{\mathsf{M}} = \{v \mid op \in \Omega^- \cup \Omega^\times \text{ acts on } v\}$ and $\widehat{\mathsf{M}}' = \{v \mid op \in \Omega^+ \cup \Omega^\times \text{ acts on } v\}$. The next two lemmas bound the number of affected vertices.

Lemma 7.4 *For any vertex $v \in \widehat{\mathsf{M}}$, $|vv^*| \in O(\mathrm{NN}_{\mathsf{M}}(v))$ and for any $v \in \widehat{\mathsf{M}}'$, $|vv^*| \in O(\mathrm{NN}_{\mathsf{M}'}(v))$.*

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 $op_v \in \Omega^- \cup \Omega^\times$ acting on v at rank r. Lemma 7.2 suggests that there exists a dependency path from a square $s \in \Sigma^\times$ to op_v. Let op_u be the operation on this path that reads s; op_u acts on a vertex u at rank r_u. By lemma 7.3, we know that $|vu| \in O(\rho^r)$. By that fact that op_u reads s, we know $|us|$ is in $O(\rho^{r_u})$ and by lemmas 3.3 and 3.4 the quadtree functions `QTAdd` and `QTRemove` guarantee that $|sv^*| \in O(|s|)$ which is in $O(\rho^{r_u})$ as well. Using the triangle inequality and the fact that $r_u \leq r$, we bound $|vv^*|$ 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 6.2 proves precisely this. ■

Lemma 7.5 (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 $|vv^*| \in [2^i, 2^{i+1})$. By lemma 7.4, 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 $\widehat{\mathsf{M}}'$. ■

8 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 8.2) and efficient (theorem 8.3).

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 the operations that become obsolete, that need to be executed, and that become inconsistent respectively; 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 `UndoOps` to remove obsolete operations and the `Dispatch` and `Fill` operations of the stable algorithm for executing fresh operations.

Figure 4 shows the pseudo-code for the `Add` and `Remove` functions for inserting and deleting a vertex v^* into and from the input, and the `PropagateWS` function for dynamic updates. Given v^*, `Add`/`Remove` 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 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, a square in $\Sigma^{\otimes}$. 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 3). The `UndoOps` function undoes the work of obsolete and fresh operations by marking all of their children for removal and by deleting quadtree dependencies (edges) from the computation graph. It also prepares the live inconsistent operations for re-execution by resetting their dependencies. The `MarkReaders` function expands the set of inconsistent operations as the set of vertices in a square changes due to removed or freshly executed fill operations.

As their notation suggests, the obsolete, fresh, and inconsistent operations used by the algorithm correspond to those defined in the stability analysis; lemma 8.1 makes this correspondence precise.

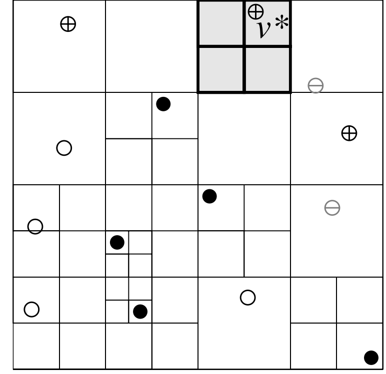

Figure 5: Dynamic update after insertion of v^*. Solid vertices are input (N), vertices marked $+$ are inserted, vertices marked $-$ are deleted. Gray squares are inconsistent. The four smaller gray squares are fresh; they replace the bigger obsolete square.

Lemma 8.1 *The set of operations processed in the dynamic update algorithm, $\Omega^{\ominus} \cup \Omega^{\oplus} \cup \Omega^{\otimes}$, is equal to 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 $B \not\subset A$ and let op be the earliest operation in $B \setminus A$. If $op \in \Omega^{-}$ then either op is a dispatch operation acting on an input vertex or there is another operation $op' \in \Omega^{-} \cup \Omega^{\times}$ that creates op. In the first case, op depends on a square in $\Sigma^{\times}$, which implies $op \in A$. In the second case, by the minimality of op, $op' \in A$. Since the update algorithm processes all children of op', $op \in A$. Similar arguments show that $op \in \Omega^{+}$ implies $op \in A$. Therefore op must be in $\Omega^{\times}$, i.e., there exists a dependency path from a square $s \in \Sigma^{\times}$ to op. Pick the longest dependency path that reaches op and let $op' \neq op$ be the latest operation on that path. If no such op' exists then op is a dispatch operation acting on an input vertex that reads a square from $\Sigma^{\times}$. The initialization in `PropagateWS` puts op in A. In the other case that op' exists, by minimality of op, op' is in A and the dependency path from op' to op ensures that our update algorithm schedules op to one of the sets $\Omega^{\ominus}$, $\Omega^{\oplus}$, or $\Omega^{\otimes}$, depending on the type of dependency between op and op'. Contradiction.

For the other direction, similarly assume the contrary and let op be the earliest operation in $A \setminus B$. If $op \in \Omega^{\ominus}$ then either op is a dispatch operation acting on an input vertex or there is another operation $op' \in \Omega^{\ominus} \cup \Omega^{\otimes}$ that creates op. In the first case, op depends on a square in $\Sigma^{\times}$, which implies $op \in B$. In the second case, by minimality of op, $op' \in B$. If $op' \in \Omega^{\times}$ then by the definition of dependency paths, we get $op \in B$. Otherwise, $op' \in \Omega^{-}$ and any operation that op' creates, more specifically op, cannot be matched by the matching μ, hence $op \in B$. Similar arguments show that $op \in \Omega^{\oplus}$ implies $op \in B$. Therefore op must be in $\Omega^{\otimes}$, i.e., op reads a square s for which the algorithm calls the function `MarkReaders` (s, t) with a smaller time t than the time of op. If $s \in \Sigma^{\otimes}$ then clearly $op \in B$; otherwise, there is another operation op' running at time t that writes into s. Again, by minimality of op, $op' \in B$ and there exists a dependency path from op' to op which puts op in B by lemma 7.2. 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 8.2 (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 inductively. We define the sets of operations according to their creation times: $\Omega_t^{\ominus} = \{op \in \Omega^{\ominus} \mid op \text{ is created at time} < t\}$ ($\Omega_0^{\ominus}$ is the set of dispatch operations acting on input vertices). 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.

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 8.3 *The `Add` and `Remove` 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 8.2, 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 5.4 applies. As discussed in section 3, the quadtree can be updated in $O(\log \Delta)$ time. Also, lemma 8.1 relates the runtime of the update algorithm to the distance between the executions with the old and new inputs. Finally, lemma 7.5 bounds the runtime of `PropagateWS` as desired. ■

9 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.

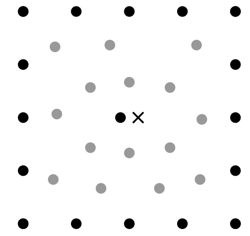

Figure 6: Inserting x creates $\Omega(\log \Delta)$ fresh Steiner vertices.

We define a gadget (see figure 6) consisting of points in the hypercube $[0, k^{-1/d}]^d$. Consider two vertices at distance $1/\Delta$ 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 [13, p.79]. The input N consists of tiling $[0, 1]^d$ with the gadgets, $k^{1/d}$ for each dimension, without any dynamic vertex. The dynamic modification sequence consists of inserting k dynamic vertices, one for each gadget.

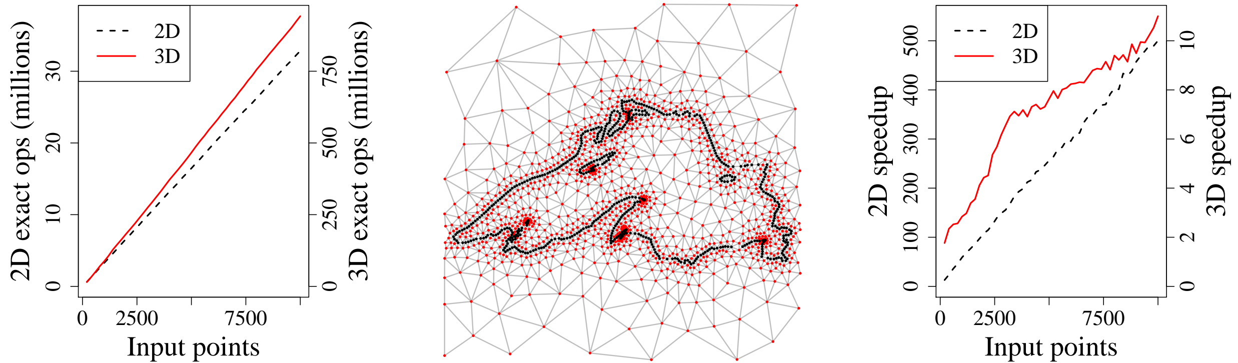

Figure 7: Left: cost of `StableWS` on random inputs. Center: A model of Lake Superior meshed by `StableWS`. Right: speedup of `PropagateWS` one unit changes relative to `StableWS` from scratch.

Lemma 9.1 *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 $\ell = \frac{1}{4} - \frac{\delta}{2}$. 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, $|v_1 x| \leq 2\rho\,\mathrm{NN}_{\mathsf{N}'}(x) = 2\rho\delta$. Also, the nearest neighbour distance of v_1 is at most $|v_1 x|$. We can use the same argument to get $|v_1 v_2| \leq 2\rho |v_1 x|$ and repeat. In other words, distance from x grows only geometrically as we walk down the segment: covering the distance ℓ requires $\Omega(\log 1/\delta) = \Omega(\log \Delta)$ many Steiner vertices. This implies that M differs from M' in at least $O(\log \Delta)$ vertices. ■

Theorem 9.2 (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, let $k = n$. Then, 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 [13]. 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. ■

10 Implementation & Experiments

We implemented[1] the `StableWS` and `PropagateWS` algorithms in C++. Given a set of vertices N, `StableWS` computes a well-spaced superset M of N and `PropagateWS` updates the output dynamically as the input is modified by insertions and deletions. Our implementation is a preliminary prototype: it follows closely the algorithmic description with minor optimizations. As with other meshing software (e.g., [26]), ours is highly susceptible to numerical error. We therefore use an exact arithmetic package based on floating-point filters which is functional and reasonably fast, but is nonetheless far from being the fastest available. We have verified the correctness of our implementation by considering numerous randomly generated inputs and some real models.

[1] Source code is available for download at http://nagoya.uchicago.edu/∼cotter/projects/wsp

Application	d	Input size	# Operations in Millions			# Operations per vertex	
			SVR	StableWS	PropagateWS	SVR	StableWS
New Zealand	2	18595	56	115	0.248	403	1190
Cape Cod	2	20930	47	99	0.234	423	1170
Lake Superior	2	33487	90	188	0.303	419	1190
SF Bay	2	85910	191	393	0.239	425	1170
Bunny	3	35947	1090	3220	307	5330	22500
Armadillo	3	172974	4380	13400	572	5460	22600

Table 1: Operation counts for SVR and StableWS, and for unit changes with PropagateWS.

In all experiments, we chose $\rho = \sqrt{2}$, and $\beta = 2$ in 2D or $\beta = 2\sqrt{2}/\sqrt{3}$ in 3D, with the color parameters $\ell(r) = \rho^{r-1/2}/\sqrt{d}$ and $\kappa = \lceil 1 + 3\sqrt{d}\beta\rho^{3/2} \rceil$. For both two and three dimensions κ is 16; the number of colors in 2D is $16^2 = 256$ and in 3D it is $16^3 = 4096$.

10.1 Synthetic Data

In these experiments, we generate point sets of double-precision floating-point numbers drawn uniformly at random from the unit box in 2D and 3D. For a given input, we measure the cost of running StableWS on the entire input, and the average cost of performing an update after a *unit dynamic change* that removes a random input vertex, updates the output using PropagateWS, adds a new vertex, and updates again. To focus on algorithmic concerns we use exact arithmetic operation counts to measure run-time cost. These dominate runtime even in highly optimized implementations.

Figure 7 shows the speedup of dynamic updates calculated as the ratio of the cost of running StableWS to the average cost of one dynamic update with PropagateWS. Each plotted point is the average over 100 different unit dynamic changes on each of 10 random inputs. We include 2D and 3D measurements on the same plot; note that the y-axis scales are different (the constant factors are larger in 3D). Consistent with our analysis, the measurements indicate that in both 2D and 3D the cost of StableWS grows close to linearly with the input sizes, while dynamic updates yield linear speedups.

10.2 Real Data

This second round of experiments was performed using several real-world models (e.g., from the Stanford 3D scanning repository), on which we compare the performances of StableWS and PropagateWS (the latter, again, with unit changes) to one of the fastest available well-spaced superset implementations, SVR [4]. We use a version of SVR that has been modified to use the same quality criteria as our algorithms, and which tracks and outputs exact arithmetic operation counts. Depending on other parameter settings, the algorithms can generate outputs of slightly different sizes (the variance is often less than 50%). We therefore provide the cost per output vertex, which offers a better basis of comparison. Table 1 shows our measurements. For each output vertex, our prototype of StableWS performs at most four times as many operations as SVR; cumulative cost measures are consistent with these results. Dynamic updates are at least two orders of magnitude faster than SVR in 2D, and still provide a large benefit in 3D.

11 Conclusion

We present a dynamic algorithm for computing a well-spaced point set 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 to these results 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] Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[2] Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In *Proceedings of the ACM SIGPLAN Conference on PLDI*, 2006.

[3] Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Duru Türkoğlu. Robust Kinetic Convex Hulls in 3D. In *16th Annual European Symposium on Algorithms*, 2008.

[4] Umut A. Acar, Benoît Hudson, Gary L. Miller, and Todd Phillips. SVR: Practical engineering of a fast 3D meshing algorithm. In *International Meshing Roundtable*, pages 45–62, 2007.

[5] Marshall Bern, David Eppstein, and John R. Gilbert. Provably Good Mesh Generation. *Journal of Computer and System Sciences*, 48(3):384–409, 1994.

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

[7] Siu-Wing Cheng, Tamal Krishna Dey, Herbert Edelsbrunner, Michael A. Facello, and Shang-Hua Teng. Sliver Exudation. *Journal of the ACM*, 47(5):883–904, 2000.

[8] Nuttapong Chentanez, Ron Alterovitz, Daniel Ritchie, Lita Cho, Kris K. Hauser, Ken Goldberg, Jonathan R. Shewchuk, and James F. O'Brien. Interactive simulation of surgical needle insertion and steering. In *Proceedings of ACM SIGGRAPH*, Aug 2009.

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

[10] Kenneth L. Clarkson, Kurt Mehlhorn, and Raimund Seidel. Four results on randomized incremental constructions. *Computational Geometry Theory and Application*, 3:185–212, 1993.

[11] Narcis Coll, Marité Guerrieri, and J. Antoni Sellarès. Mesh modification under local domain changes. In *15th Intl. Meshing Roundtable*, pages 39–56, 2006.

[12] Sariel Har-Peled and Alper Üngör. A time-optimal Delaunay refinement algorithm in two dimensions. In *21st Symposium on Computational Geometry*, pages 228–236, 2005.

[13] Benoît Hudson. *Dynamic Mesh Refinement*. PhD thesis, School of Computer Science, Carnegie Mellon University, December 2007. Available as Technical Report CMU-CS-07-162.

[14] Benoît Hudson, Gary L. Miller, and Todd Phillips. Sparse Voronoi Refinement. In *15th Intl. Meshing Roundtable*, pages 339–356, 2006. Long version in Carnegie Mellon University Tech. Report CMU-CS-06-132.

[15] Benoît Hudson and Duru Türkoğlu. An efficient query structure for mesh refinement. In *Canadian Conf. on Comp. Geometry*, 2008.

[16] Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, 2009.

[17] Xiang-Yang Li and Shang-Hua Teng. Generating well-shaped Delaunay meshes in 3D. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 28–37, 2001.

[18] Xiang-Yang Li, Shang-Hua Teng, and Alper Üngör. Simultaneous refinement and coarsening for adaptive meshing. *Engineering with Computers*, 15(3):280–291, 1999.

[19] Gary L. Miller, Dafna Talmor, Shang-Hua Teng, Noel Walkington, and Han Wang. Control Volume Meshes Using Sphere Packing: Generation, Refinement and Coarsening. In *5th Intl. Meshing Roundtable*, pages 47–61, 1996.

[20] Neil Molino, Zhaosheng Bao, and Ron Fedkiw. A virtual node algorithm for changing mesh topology during simulation. In *SIGGRAPH*, 2004.

[21] Doug Moore. The cost of balancing generalized quadtrees. In *SMA '95: Proceedings of the third ACM symposium on Solid modeling and applications*, pages 305–312, New York, NY, USA, 1995. ACM.

[22] Ketan Mulmuley. Randomized multidimensional search trees (extended abstract): dynamic sampling. In *Proceedings of the 7th Annual Symposium on Computational Geometry*, pages 121–131, 1991.

[23] Han-Wen Nienhuys and A. Frank van der Stappen. A Delaunay approach to interactive cutting in triangulated surfaces. In *fifth Intl. Workshop on Algorithmic Foundations of Robotics*, 2004.

[24] Jim Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995.

[25] Otfried Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 197–206, 1991.

[26] Jonathan Richard Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3):305–363, 1997.

[27] Daniel Spielman, Shang-Hua Teng, and Alper Üngör. Parallel Delaunay refinement: Algorithms and analyses. *IJCGA*, 17:1–30, 2007.

[28] Dafna Talmor. *Well-Spaced Points for Numerical Methods*. PhD thesis, Carnegie Mellon University, August 1997. Available as Technical Report CMU-CS-97-164.