Dynamic Well-Spaced Point Sets

Umut A. Acar* Andrew Cotter* Benoît Hudson* Duru Türkoğlu†

December 8, 2009

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 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 is bounded by $O(\log n)$ 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 by showing that there exists inputs and modifications that require $\Omega(\log \Delta)$ Steiner points to be inserted to the output. 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.

*Toyota Technological Institute at Chicago
†University of Chicago

1 Introduction

We call a set of points M *well-spaced* if the Voronoi cell of each point has a good aspect ratio, i.e., the ratio of the distance to the farthest point in the Voronoi cell divided by the distance to the nearest neighbor in M is small [Tal97]. 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 [LT01, CDE$^+$00]. The Voronoi diagram of a well-spaced point set is also immediately useful for the control volume method [MTTW95].

Given a finite set of points N in the d-dimensional unit hypercube, $[0,1]^d$, the well-spaced point set problem is to construct an output $\mathsf{M} \supseteq \mathsf{N}$ that is well-spaced. 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, N. This problem has been studied since the late 1980s (e.g. [Che89, BEG94, Rup95]), with several recent results obtaining fast runtimes [HPÜ05, HMP06, STÜ07, HT08]. 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 of the output 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., [NvdS04]) or they are asymptotically no faster than running a static algorithm from scratch [LTU99, MBF04, CGS06].

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 $\frac{1}{\delta}$, where δ is the distance between the closest pair of points in the larger input. If the spread is polynomially bounded in the size of the input n, 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 optimal-sized, well-spaced supersets (Section 3). To enable dynamization, in addition to the output, the algorithm constructs a *computation graph* that represents the operations performed during the execution and the dependences between them. A key property of this algorithm is that it is *stable* in the sense that it produces similar computation graphs with similar inputs, e.g., that differ by one point. We make this property precise by describing a *distance* measure between computations graphs and bounding it by $O(\log \Delta)$ when inputs differ by a single point (Lemma 6.4). Informally, the distance measure corresponds to the number of output points that are *affected* by an insertion/deletion of a point into/from the input.

Taking advantage of stability, we provide a *dynamic update algorithm* (Section 7) that updates the output and the computation graph in time proportional the distance between them, i.e., in $O(\log \Delta)$ time for a single insertion/deletion. The dynamic update algorithm achieves fast updates by identifying the operations that are affected by the modification to the input and deleting and re-executing them as necessary. At a high level, the approach can be viewed as a dynamization technique, which has been used effectively for a relatively broad range of algorithms (e.g. [Mul91, Sch91, BDS$^+$92, CMS93]). 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 7.2). Consequently, the output remains both well-spaced and size-optimal with respect to the modified input (Theorem 7.3).

The run-time of the dynamic-update algorithm directly depends on the similarity between computations as measured by the number of affected output points. We make sure that this quantity is small by carefully designing our stable algorithm to maintain 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 [STÜ07]. Second, we pick Steiner points by making local decisions only, using clipped Voronoi cells [HT08]. These techniques enable us to process each point only once and help isolate and limit the effects of a modification. Our proof follows from a spacing-and-packing argument. The spacing argument shows that any affected point has an empty ball around itself whose radius is proportional to its distance to the dynamic point v^*. Intuitively, the further away the affected points are from v^*, the further away they are from each other. We then apply a packing argument to show that there can be only a constant number of affected points at each level, consequently, $O(\log \Delta)$ in total.

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

2 Preliminaries

We present several definitions and the notation used in the rest of the paper; Figure 1 illustrates some of them. We work within the d-dimensional unit hypercube $[0,1]^d$. In the static setting, our algorithm takes as input a set of vertices, N, and produces as output a well-spaced superset M. For clarity, we use the term *point* to refer to any point in space and the term *vertex* to refer to the input and output points. Given a vertex set $\mathcal{M}$, 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 $x \in [0,1]^d$ such that for all $u \in \mathcal{M}$, $|vx| \leq |ux|$. The *outradius* of the Voronoi cell of v is the distance from v to the farthest point in $\mathrm{Vor}_{\mathcal{M}}(v)$ and the *aspect ratio* of $\mathrm{Vor}_{\mathcal{M}}(v)$ is its outradius divided by $\mathrm{NN}_{\mathcal{M}}(v)$. Following Talmor [Tal97], we say that a vertex is *ρ-well-spaced* if the aspect ratio of its Voronoi cell is bounded by ρ. We say that $\mathcal{M}$ is ρ-well-spaced if every vertex in $\mathcal{M}$ is ρ-well-spaced. We define the *β-clipped Voronoi cell*, written $\mathrm{Vor}^{\beta}_{\mathcal{M}}(v)$, as the intersection of $\mathrm{Vor}_{\mathcal{M}}(v)$ with the ball of radius $\beta\,\mathrm{NN}_{\mathcal{M}}(v)$ centered at v [HT08]. Note that a point v is ρ-well-spaced if and only if $\mathrm{Vor}^{\rho}_{\mathcal{M}}(v) = \mathrm{Vor}_{\mathcal{M}}(v)$.

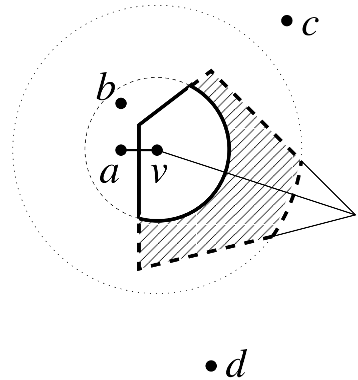

Figure 1: $\mathcal{M} = \{a, b, c, d, v\}$. $\mathrm{NN}_{\mathcal{M}}(v) = |va|$. Thick solid and dashed boundaries depict $\mathrm{Vor}^{\rho}_{\mathcal{M}}(v)$ and $\mathrm{Vor}^{\beta}_{\mathcal{M}}(v)$.

We define *local feature size* of a point $x \in [0,1]^d$, written $\mathrm{lfs}(x)$, as the distance from x to the second-nearest vertex of N. We say that a set of vertices $\mathcal{M}$ is *size-conforming*, if $\mathrm{NN}_{\mathcal{M}}(v) \in \Omega(\mathrm{lfs}(v))$ for all $v \in \mathcal{M}$. Our algorithm guarantees that the output is size-conforming. Using this property, we prove the optimality result on the output size which directly affects the runtime bound of our static algorithm.

Our algorithm uses a point location structure based on the balanced quadtree of Bern, Eppstein, and Gilbert [BEG94] (Appendix A). It is relatively straightforward to dynamize the balanced quadtree and to extend it to d dimensions (we use 'quadtree' and 'quadtree node' to mean 2^d-tree and d-hypercube). Externally, we use only the leaves of the quadtree, which we refer to as *squares*. The quadtree squares store neighbor pointers and a list of the vertices they contain, to support fast searches. Vertices store the square that contains them, avoiding the need to search through the tree structure. The quadtree supports the functions `QTBuild`, `QTAdd`, `QTRemove`, `QTApxNN`, and `QTClippedVoronoi`. Function `QTBuild(N)`

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

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

ClippedVoronoi (op)
  v ← op.vertex
  (CV, S) ← QTClippedVoronoi(v, β)
  foreach s ∈ S do s.access ← s.access ∪ {op}
  op.reads ← S
  return (v, CV, S)
```

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

Dispatch (op, Ω) =
  (v, CV, S) ← ClippedVoronoi(op)
  r ← ⌊log_ρ NN(v)⌋ (via CV)
  if r ≥ op.rank then
    Ω ← Ω ∪ {NewOp(v, op, r, Color(v, r))}
  foreach neighbor u of v (via CV) do
    r_u ← ⌊log_ρ |uv|⌋
    if r_u ≥ op.rank then
      Ω ← Ω ∪ {NewOp(u, op, r_u, Color(u, r_u))}

Fill (op, Ω) =
  (v, CV, S) ← ClippedVoronoi(op)
  while v is not ρ-well-spaced do
    choose u ∈ CV such that |uv| ≥ ρ NN(v)
    s ← square of S that contains u
    u.square ← s
    s.vertices ← s.vertices ∪ {u}
    op.steiners ← op.steiners ∪ {u}
    Ω ← Ω ∪ {NewOp(u, op, ⌊log_ρ |uv|⌋ , 0)}
    update CV with u
```

Figure 2: The pseudo-code for our stable algorithm.

constructs a quadtree for the set of vertices N in $O(n \log \Delta)$ time and returns the quadtree. 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 and the set of squares that are deleted or that become internal quadtree nodes. Function $\texttt{QTApxNN}(v)$ returns a value in $\Theta(\mathrm{NN}_{\mathsf{N}}(v))$ in $O(1)$ time and $\texttt{QTClippedVoronoi}(v, \beta)$ returns $\mathrm{Vor}^{\beta}_{\mathcal{M}}(v)$ and the set of squares it reads in $O(1)$ time under certain assumptions [HT08] that our algorithm meets.

3 A Stable Algorithm

Given a set of inputs points N the algorithm starts by constructing a quad-tree and stores it in the variable Π for use in dynamic updates (PropagateWS) . The algorithm computes the output by creating and performing two kinds of operations, *dispatch* and *fill*, in a carefully controlled schedule to ensure efficiency and stability; the operations are stored in the variable Ω. NewOp creates an operation, initializing its fields: rank, color, the vertex that we say it *acts on*, a list of Steiner vertices that it creates (if any), a list of squares that it reads, and pointers to its children, i.e., operations that it creates. Except for some initial operations, the *rank* of an operation acting on a vertex v is the floor of the logarithm base ρ of the nearest neighbor distance of v at the time of the operation's creation. We define a color for each operation in order to limit the dependences between operations of the same rank: two operations that share the same rank and color are mutually independent (a fact used by Lemma 6.2). The *color* of a dispatch operation acting on a vertex v is zero. For fill operations the color is the coordinates of v rounded to a periodic square lattice with scale $\rho^r/\sqrt{d}$ and period κ (see Appendix B). The rank and the color of an operation defines its *time*, ordered by the natural lexicographic ordering on pairs, e.g., $(r, 0) < (r, 1) < \ldots < (r + 1, 0)$.

The algorithm starts by creating a quadtree and then creates a dispatch operation for each input vertex using a constant factor approximation of their actual rank as computed by QTApxNN. It then executes the operations in the order of their time by considering the $\Theta(\log \Delta)$ ranks and the $\kappa^d + 1$ colors. In the pseudo-

code, we use $\Omega|_{r,c}$ to refer to the operations with time (r, c). If an operation *op* acting on vertex v has color zero, we apply function `Dispatch`, which creates a fill operation for v and for each vertex that is a neighbor of v's β-clipped Voronoi cell. If *op* has non-zero color, we apply function `Fill`, which checks whether v is ρ-well-spaced by finding its β-clipped Voronoi cell. If v is ρ-well-spaced then the function returns, otherwise, it creates Steiner vertices until v becomes ρ-well spaced, adding each created Steiner vertex to the appropriate square and scheduling a dispatch operation for it.

Output and Time. We use M to refer to the set of output vertices (input and Steiner vertices). In the analysis of the algorithm, 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 equal to the input, N, and M_∞ is equal to the final output, M. For readability, we use t instead of M_t in the subscript, e.g., NN_t instead of $\mathrm{NN}_{\mathsf{M}_t}$.

Computation Graph. When executed, our algorithm creates a representation of the execution, which we call a *computation graph* $G = (V, E)$. The nodes, $V = \Sigma \cup \Omega$, consist of the set of squares (Σ) and the set of operations (Ω) at the completion of the algorithm. The edges represent various dependences between operations and squares. If an operation *op* creates another operation *op*$'$ then (op, op') becomes an edge ($(op, op') \in E$, recorded by storing *op*$'$ in the `children` field of *op*). If an operation *op* reads a square s via the `QTClippedVoronoi` operation then (s, op) becomes an edge (recorded by storing *op* in s`.access` and s in *op*`.reads`). Finally, if an operation *op* writes a square s by inserting a new Steiner vertex u into it then (op, s) becomes an edge (recorded by storing u in the `steiners` field of *op*, u points to s via its `square` field). We consider each edge to be tagged with the time (rank and color) of the operation that creates it, e.g., in the description above this is the time of *op*.

4 Output Quality and Size

We prove that our algorithm guarantees that the output, M, is ρ-well-spaced and size-optimal with respect to N. We prove size-optimality by showing that M is size-conforming using a technique introduced by Ruppert [Rup95]; due to space restrictions we leave this proof to Appendix C. For ρ-well-spacedness, we establish the key invariant that after running `Fill` on a vertex (i.e., making it well spaced), we never need to visit it again—we incrementally progress towards a ρ-well-spaced output.

Lemma 4.1 *For all t, M_t is size-conforming. Consequently, M is size-optimal with respect to N.*

Lemma 4.2 *Let v be a vertex in $\mathcal{M}$ such that every vertex $u \in \mathcal{M}$ with $\mathrm{NN}_{\mathcal{M}}(u) \leq \mathrm{NN}_{\mathcal{M}}(v)/\rho$ is ρ-well-spaced. Then no fill operation can create a new nearest neighbor for v.*

Proof: Assume that the fill operation acts on a vertex w. If w is ρ-well-spaced in $\mathcal{M}$, the operation does not create any Steiner vertices and the lemma holds trivially. Otherwise, the operation inserts a vertex w'. Our algorithm chooses Steiner vertices in $\mathrm{Vor}_{\mathcal{M}}(w)$—in particular, $|vw'| \geq |ww'|$—at distance $|ww'| \geq \rho\,\mathrm{NN}_{\mathcal{M}}(w)$. Given that w is not ρ-well-spaced, $\rho\,\mathrm{NN}_{\mathcal{M}}(w) > \mathrm{NN}_{\mathcal{M}}(v)$. Unwinding the inequalities, we get $|vw'| > \mathrm{NN}_{\mathcal{M}}(v)$. That is, w' cannot be a new nearest neighbor of v. ■

Lemma 4.3 (Progress) *At time $t = (r, 0)$, every vertex $v \in \mathsf{M}_t$ with $\mathrm{NN}_t(v) < \rho^r$ is ρ-well-spaced.*

Proof: At the minimum rank, there are no vertices with smaller nearest neighbor distance, so the claim is trivially true. Assume that the lemma is true up to rank r, that is, for $t = (r, 0)$, every vertex $u \in \mathsf{M}_t$ with $\mathrm{NN}_t(u) < \rho^r$ is ρ-well-spaced. We want to show that our claim holds for $t' = (r + 1, 0)$. Consider a vertex $v \in \mathsf{M}_{t'}$ with $\mathrm{NN}_{t'}(v) < \rho^{r+1}$. All Steiner vertices inserted at rank r have their nearest neighbors at distance at least ρ^{r+1}. This implies that v is not a Steiner vertex inserted at rank r; so $\mathrm{NN}_t(v)$ is defined and $< \rho^{r+1}$ as well. We know that if $\mathrm{NN}_t(v) \geq \rho^r$, there exists a fill operation that acts on v at rank r (Lemma C.1). After executing that operation, v becomes ρ-well-spaced. Also, Lemma 4.2 shows that all ρ-well-spaced vertices $u \in \mathsf{M}_t$ with $\mathrm{NN}_t(u) < \rho^{r+1}$ remain ρ-well-spaced. Therefore, our claim holds. ■

Theorem 4.4 *The static algorithm constructs a size-optimal ρ-well-spaced superset M of its input N.*
Proof: That M is ρ-well-spaced follows from the Progress Lemma and the fact that our algorithm iterates over all ranks. Lemma 4.1 proves the size bound. ■

5 Runtime

We analyze the running time of our static algorithm and emphasize two lemmas that turn out to be useful in the analysis of our dynamic algorithm. The first lemma (Lemma 5.1) proves that throughout the algorithm, the nearest neighbor distance of a vertex v changes only by a constant factor. The second (Lemma 5.2) 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. Due to space restrictions, we leave most of the proofs to Appendix D.

Lemma 5.1 *Let t be the time at which v is created ($t = 0$ for input vertices). Then $\mathrm{NN}_t(v) \in \Theta(\mathrm{NN}_\infty(v))$.*

Lemma 5.2 *If an operation that acts on v runs at rank r, then $\mathrm{NN}_\infty(v) \in \Theta(\rho^r)$.*

Lemma 5.3 *Every operation runs in $O(1)$ time.*
Proof: Pick an operation that acts 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 5.2 and 5.1 together show that $\mathrm{NN}_t(v) \in \Theta(\rho^r)$. Hudson and Türkoğlu [HT08] show that these are sufficient conditions to guarantee `QTClippedVoronoi` runs in constant time.

The dispatch operation loops as many times as there are clipped Voronoi neighbors. Since `QTClipped-Voronoi` runs in constant time, there can be only $O(1)$ neighbors. The fill operation has a loop that adds vertices until v is ρ-well-spaced. Each vertex u is chosen in $\mathrm{Vor}_t^\beta(v)$ but far from v: $\mathrm{NN}_t(u) = |uv| \geq \rho\, \mathrm{NN}_t(v)$. Therefore, we can associate non-overlapping empty balls of radius $\rho\, \mathrm{NN}_t(v)/2$ around every Steiner vertex. Since all of the Steiner vertices are in a ball of radius $\beta\, \mathrm{NN}_t(v)$ around v, a packing argument shows that there are $O(1)$ Steiner vertices created for each fill operation. ■

Lemma 5.4 *For every vertex $v \in \mathsf{M}$, there are $O(1)$ operations that act on v.*

Theorem 5.5 *The static algorithm runs in $O(n \log \Delta)$ time.*
Proof: Building the quadtree using `QTBuild` takes $O(n \log \Delta)$ time. There are a constant number of operations that act on a vertex in the output and each operation takes constant time. Thus, the total runtime is $O(n \log \Delta + m)$, where $m = |M|$. That $m \in O(n \log \Delta)$ follows from our dynamic runtime bound. ■

6 Dynamic Stability

We call two inputs N and N' *related* if they differ by one vertex, i.e., one can be obtained from the other by inserting or deleting a vertex. To analyze the stability of algorithm `StableWS`, we define a notion of distance between its executions with related inputs and prove that this distance is bounded by $O(\log \Delta)$ in the worst-case, where Δ is the larger geometric spread among the two inputs N and N' (Lemma 6.4).

As described in Section 3, `StableWS(N)` creates $G = (V, E)$ by building a quadtree Σ and a set of operations Ω, $V = \Sigma \cup \Omega$, and the edges E are defined by the pointer structure that is constructed. Similarly define V', E', Σ', and Ω' for `StableWS(N')`. We say that two squares $s \in \Sigma$ and $s' \in \Sigma'$ are *identical*, written $s \equiv s'$, if s and s' have the same geometry, i.e., the same corner points and size. We say that 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. We define a *matching* $\mu : V \to V'$ between G and G' as $\mu = \mu_o \cup \mu_s$, where μ_o is the largest set satisfying $\mu_o = \{(op, op') \mid op \in \Omega \wedge op' \in \Omega' \wedge (op \equiv op') \wedge (\text{parent}(op), \text{parent}(op')) \in \mu_o\}$ and $\mu_s = \{(s, s') \mid s \in \Sigma \wedge s' \in \Sigma' \wedge s \equiv s'\}$. Informally, the matching pairs squares of G with the identical squares of G' and pairs the operations of G with the identical operations of G' as long as their parents (the operations that create them, if any) are also paired. We use $\mathsf{dom}(\mu)$ and $\mathsf{range}(\mu)$ to refer to the domain and the range of μ. We say that nodes $u \in V$ and $u' \in V'$ *match* if $\mu(u) = u'$.

Given $G = (V, E)$ and $G' = (V', E')$ and their matching μ, let the matching $\mu' = \mu \cup \{(u, u) \mid u \in V' \setminus \mathsf{range}(\mu)\}$ be a surjective matching onto the nodes V' of G. We combine the computation graphs in a *union graph* as follows: $G^{\cup} = (V \cup \mu'(V'), E \cup \mu'(E'))$, 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 the union graph, we say that a path $(u_0, u_1, \ldots, u_h)$ is a *dependence path* if the times of the edges $(u_0, u_1), (u_1, u_2), \ldots, (u_{h-1}, u_h)$ increase monotonically.

We partition the nodes of the union graph $G^{\cup} = (V^{\cup}, E^{\cup})$ into several categories. The nodes $V^- = V \setminus \mathsf{dom}(\mu)$ are called the *obsolete* nodes (squares Σ^-, operations Ω^-); these are the nodes of G that have no matching pairs in G'. The nodes $V^+ = V' \setminus \mathsf{range}(\mu)$ are called *fresh* nodes (squares Σ^+, operations Ω^+); these are the nodes of G' that have no matching pairs in G. Additionally, we call a square $s \in V^{\cup}$ *inconsistent* if it is fresh or obsolete, or if it contains the vertex v^*. We say that an operation $op \in \mathsf{dom}(\mu)$ is inconsistent if it is reachable from an inconsistent square via a dependence path. We represent inconsistent nodes with $V^{\times}$ (squares $\Sigma^{\times}$, operations $\Omega^{\times}$). We define the distance between the execution of two related inputs N and N' with graphs G and G' as the number of obsolete, fresh, or inconsistent operations of the union graph, i.e., $\text{distance}(\texttt{StableWS}(\mathsf{N}), \texttt{StableWS}(\mathsf{N}')) = |\Omega^- \cup \Omega^+ \cup \Omega^{\times}|$.

Lemma 6.1 *For every operation in $\Omega^- \cup \Omega^+ \cup \Omega^{\times}$, there exists a dependence path from a square in $\Sigma^{\times}$.*

As proven in [HT08], `QTClippedVoronoi` satisfies the following property: given a size-conforming set of vertices $\mathcal{M}$ and a square s read by `QTClippedVoronoi`(v, β), for all $x \in s$, $|vx| \in O(\text{NN}_{\mathcal{M}}(v))$. By using this locality property, we relate the operations on a dependence path geometrically.

Lemma 6.2 *Consider two operations op_1 and op_2 in $G^{\cup}$ acting on vertices v_1 and v_2. If there exists a dependence path from op_1 to op_2 and op_2 is at rank r, then $|v_1 v_2| \in O(\rho^r)$.*

Proof: First, we show that for any edge, 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. The distance between two operations is the distance between the vertices on which they act. Without loss of generality, pick an edge $e \in E$ with time $t_e = (r_e, c_e)$. The edge e consists of an operation

$op \in \Omega$ acting on v at time t_e and either a square s that it accesses (reads/writes) or another operation op' that it schedules. Then, using the locality result from [HT08], we bound the distance between op and s by $O(\mathrm{NN}_{t_e}(v))$. Also, op' is within the same distance. Lemmas 5.1 and 5.2 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.

Following the definition of dependence paths, the times of the edges on the path from op_1 to op_2 monotonically increase. Then, assuming that the rank of op_1 is r', there can be at most κ^d many edges for each rank between r' and r. Therefore, in the worst case, the distance between v_1 and v_2 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, $|v_1v_2| \in O(\rho^r)$. ■

In order to bound the distance between two executions, we focus on the vertices rather than the operations. We say that a vertex is *affected* if there exists an unmatched or an inconsistent operation that acts on it. Since there are a constant number of operations acting on a given vertex (Lemma 5.4), the number of affected vertices exactly measures the distance between two executions. We define the sets of affected vertices in both executions: $\hat{\mathsf{M}} = \{op.\texttt{vertex} \mid op \in \Omega^- \cup \Omega^\times\}$ and $\hat{\mathsf{M}}' = \{op.\texttt{vertex} \mid op \in \Omega^+ \cup \Omega^\times\}$. The following two lemmas demonstrate the spacing and packing arguments respectively.

Lemma 6.3 *For any vertex $v \in \hat{\mathsf{M}}$, $|vv^*| \in O(\mathrm{NN}_{\mathsf{M}}(v))$ and for any $v \in \hat{\mathsf{M}}'$, $|vv^*| \in O(\mathrm{NN}_{\mathsf{M}'}(v))$.*
Proof: We prove the lemma for $v \in \hat{\mathsf{M}}$; symmetric arguments apply for $\hat{\mathsf{M}}'$. By definition of $\hat{\mathsf{M}}$, there exists an operation $op_v \in \Omega^- \cup \Omega^\times$ acting on v at rank r. Lemma 6.1 suggests that there exists a dependence 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 6.2, 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 Lemmas A.1 and A.2 imply $|sv^*| \in O(|s|)$ which is in $O(\rho^{r_u})$ as well. Using the triangle inequality and the fact that $r_u \le r$, 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 in M. Lemma 5.2 proves precisely this. ■

Lemma 6.4 (Distance) *The distance between two related executions is bounded by $O(\log\Delta)$.*
Proof: Consider the vertices $v \in \hat{\mathsf{M}}$ with $|vv^*| \in [2^i, 2^{i+1})$. By Lemma 6.3, we can assign non-overlapping empty balls of radius $\Omega(2^i)$ to them. Therefore, there are at most a constant number of such vertices for any i. At most $O(\log\Delta)$ values of i cover $\hat{\mathsf{M}}$, so $|\hat{\mathsf{M}}| \in O(\log\Delta)$. The same argument applies to $\hat{\mathsf{M}}'$. ■

7 Dynamic Update Algorithm

We describe an algorithm for updating the output of `StableWS` dynamically when the input is modified by an insertion or deletion of a vertex, prove that it is correct (Lemma 7.2) and efficient (Theorem 7.3).

After we run `StableWS` with some input set to obtain an output and a computation graph, we can change the input dynamically and update the output and the computation graph. Figure 3 shows the pseudo-code for the `Add` and `Remove` functions for inserting and deleting a vertex v^* into and from the input (respectively), and the `PropagateWS` function for dynamic updates. Given v^*, `Add` and `Remove` update the quadtree and determine the set of inconsistent squares $\Sigma^\otimes$ and call `PropagateWS` with this set. The structure of `PropagateWS` is similar to that of `StableWS` but more involved. It maintains distinct operation sets marked for removal $\Omega^\ominus$, for execution $\Omega^\oplus$, and for re-execution $\Omega^\otimes$. As their notation suggests these are related to the obsolete, fresh, and inconsistent operations defined in the stability analysis; Lemma 7.1 makes this correspondence precise. `PropagateWS` takes the previous input set N and the inconsistent squares $\Sigma^\otimes$. It starts by updating the operation sets by finding the input vertices that are contained in these squares, deleting their dispatch operations, and creating new dispatch operations for them. The algorithm

```
Globals:  ρ, β, κ, Ω⊖, Ω⊕, Ω⊗

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

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

UndoOps (r, c) =
  foreach op ∈ (Ω⊖ ∪ Ω⊗)|_{r,c} do
    Ω⊖ ← Ω⊖ ∪ op.children
    foreach s ∈ op.reads do
      s.access ← s.access \ {op}
    foreach v ∈ op.steiners do
      s ← v.square
      s.vertices ← s.vertices \ {v}
      MarkReaders(s, (r, c))
  Ω⊗ ← Ω⊗ \ Ω⊖|_{r,c}
  ResetEdges(Ω⊗|_{r,c})

PropagateWS (N, Σ⊗) =
  foreach s ∈ Σ⊗ do
    MarkReaders(s, 0)
    foreach v ∈ s.vertices ∩ N do
      Ω⊖ ← Ω⊖ ∪ {Dispatch of v}
      r_v ← ⌊log_ρ QTApxNN(v)⌋
      Ω⊕ ← Ω⊕ ∪ {NewOp(v, nil, r_v, 0)}

  R ← min rank in Ω⊖ ∪ Ω⊕ ∪ Ω⊗
  for r = R to ⌊log_ρ √d⌋ do
    UndoOps(r, 0)
    foreach op ∈ (Ω⊕ ∪ Ω⊗)|_{r,0} do
      Dispatch(op, Ω⊕)

    for c = 1 to κ^d do
      UndoOps(r, c)
      foreach op ∈ (Ω⊕ ∪ Ω⊗)|_{r,c} do
        Fill(op, Ω⊕)
        foreach v ∈ op.steiners do
          MarkReaders(v.square, (r, c))

MarkReaders (s, t) =
  foreach op ∈ s.access do
    if (op.rank, op.color) > t then
      Ω⊗ ← Ω⊗ ∪ {op}
```

Figure 3: The pseudo-code for the dynamic algorithm.

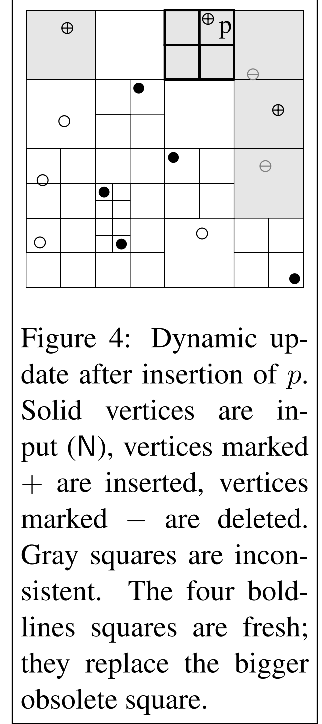

Figure 4: Dynamic update after insertion of p. Solid vertices are input (N), vertices marked $+$ are inserted, vertices marked $-$ are deleted. Gray squares are inconsistent. The four bold-lines squares are fresh; they replace the bigger obsolete square.

then proceeds in time order, first undoing the operations marked for removal and re-execution ($\Omega^{\ominus} \cup \Omega^{\otimes}$) by calling UndoOps, then performing operations marked for (re-) execution ($\Omega^{\oplus} \cup \Omega^{\otimes}$) by appropriately calling Dispatch and Fill (Figure 2). UndoOps undoes the work of the operations in $\Omega^{\ominus} \cup \Omega^{\otimes}$ by marking all of their children for removal and by deleting quadtree dependences (edges) from the computation graph. It also prepares the operations in $\Omega^{\otimes} \setminus \Omega^{\ominus}$ for re-execution by resetting dependences tracked by these operations. Dispatch and Fill appropriately insert new nodes and edges into the computation graph. When removed or executed, fill operations can update the squares causing more operations to become inconsistent, which are identified by MarkReaders.

When completed, PropagateWS updates the output (to $\tilde{\mathsf{M}}$) and the computation graph (to $\tilde{G}$) as if the algorithm StableWS is run from-scratch with the updated input which computes M' and G' (Lemma 7.2).

Lemma 7.1 *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, $\Omega^{-} \cup \Omega^{+} \cup \Omega^{\times}$.*

Lemma 7.2 (Isomorphism) *The output sets $\tilde{\mathsf{M}}$ and M' are equal and there is an isomorphism $\phi : \tilde{G} \to G'$ that preserves the vertex and time of each operation.*

Theorem 7.3 *The Add and Remove functions modify the output in $O(\log \Delta)$ time, for a well-spaced output of size within a constant factor of the optimal with respect to the updated input.*

Proof: By Lemma 7.2, we know that the output is the same as what would have been generated by running Refine from scratch with the new input; therefore, Theorem 4.4 applies. The quadtree can be updated in $O(\log \Delta)$ time. Furthermore, Lemmas 6.4 and 7.1 bound the runtime of Propagate as desired. ■

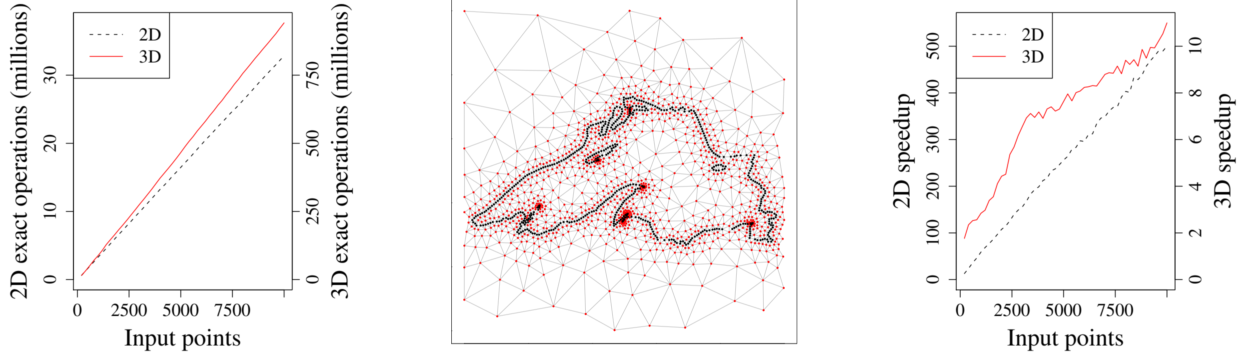

Figure 6: **Left:** cost of `StableWS` on random inputs. **Center:** a model of Lake Superior meshed by `StableWS`. **Right:** speedup of `PropagateWS` one unit changes relative to `StableWS` from scratch.

8 Lower bound

We prove a lower bound that any algorithm that explicitly maintains the well-spaced superset must take $\Omega(\log \Delta)$ 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.

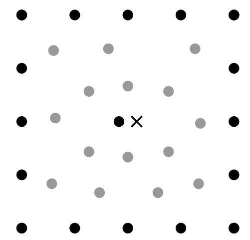

Figure 5: Inserting x creates $\Omega(\log \Delta)$ Steiner vertices.

Theorem 8.1 (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.*

9 Implementation and Experiments

We implemented `StableWS` and `PropagateWS` algorithms in C++. Given a set of vertices N, `StableWS` can be run to compute a well-spaced superset M of N, and `PropagateWS` can update the output dynamically as the input is modified by insertions and deletions. Our implementation is a preliminary prototype that closely follows the algorithmic description with relatively small optimizations. As with prior meshing software, ours is highly susceptible to numerical error. For rapid development, we used an exact arithmetic package based on floating-point filters and GMP rather than implementing Shewchuk-style adaptive predicates [She97]. This is an engineering concern independent of the choice of algorithm. Although not optimized, our implementation is reasonably well tested: we verified that it correctly yields well-spaced outputs on numerous randomly generated point sets as well as on real models. To focus on algorithmic concerns we use exact arithmetic operation counts to measure run-time costs. These dominate the runtime of even highly optimized meshing codes.

Synthetic Data. We generate point sets of double-precision floating-points numbers with varying sizes drawn uniformly at random from the unit box in two and three dimensions. For a given input, we measured the cost for running our algorithm from scratch with the input, and the average cost of a dynamic update after a *unit change*: removing a random input vertex, updating, adding a new vertex and updating again. Figure 6 shows the speedup of dynamic updates, calculated by dividing the cost of the from-scratch run (`StableWS`)

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

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

by the average cost of one dynamic update (PropagateWS). Each point is the average of performing 100 different unit 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, since the constant factors are larger in 3D. Consistent with our analysis, the measurements show that in both 2D and 3D the cost of our algorithm grows close to linearly with the input sizes, while dynamic updates yield linear speedups.

Real Data. We performed experiments with several real-world models (e.g., from Stanford 3D scanning repository) and compared the performance of StableWS and PropagateWS to the fastest available well-spaced superset code, SVR [AHMP07]. For these experiments, we use a version of SVR that uses the same quality criteria as our algorithms. Depending on other parameter settings, the algorithms can generate outputs of slightly different sizes (the variance is often less than 50%). We therefore measure the cost per output point, which offers a better basis of comparison. Table 1 shows our measurements. For each output point, our prototype of StableWS performs at most four times as many operations as SVR; cumulative cost measures are consistent with these measures. Dynamic updates are at least two orders of magnitude faster even than SVR in 2D, and still provide a large benefit in 3D.

10 Conclusion

We present a dynamic algorithm for computing a well-spaced point set of a dynamically changing set of input points. Our algorithm is time-efficient, consumes linear space, finds an optimal-size output, 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 points themselves are similar. To achieve stability (similarity of computation graphs), our stable algorithm operates on the input set by using local operations based on a notion of restricted Voronoi cell calculation and carefully schedules these operations to reduce dependencies between them by taking advantage of two key properties: 1) all computations can be partitioned into logarithmic number of sets such that only computations within each set interact (ranks), and 2) computations operating on far-enough points can be processed independently (coloring). These properties suffice to prove the desired stability results. We present a dynamic update algorithm that takes advantage of stability to update the output efficiently. Our analysis involves some tricky calculations and our asymptotic bounds have reasonably large constant factors. To assess the practicality of our approach we present a prototype implementation. Our experiments show that the algorithm can be implemented, and delivers performance consistent with our theoretical bounds competing reasonably well with state of the art mesher and delivering asymptotic speedups. We expect a well-polished implementation will provide static performance directly comparable to the current state of the art, and dynamic performance orders of magnitude speedups.

References

[AHMP07] 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.

[BDS+92] 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.

[BEG94] Marshall Bern, David Eppstein, and John R. Gilbert. Provably Good Mesh Generation. *Journal of Computer and System Sciences*, 48(3):384–409, 1994.

[CDE+00] 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.

[CGS06] Narcis Coll, Marité Guerrieri, and J. Antoni Sellarès. Mesh modification under local domain changes. In *15th International Meshing Roundtable*, pages 39–56, 2006.

[Che89] L. Paul Chew. Guaranteed-quality triangular meshes. Technical Report TR-89-983, Department of Computer Science, Cornell University, 1989.

[CMS93] Kenneth L. Clarkson, Kurt Mehlhorn, and Raimund Seidel. Four results on randomized incremental constructions. *Computational Geometry Theory and Application*, 3:185–212, 1993.

[HMP06] Benoît Hudson, Gary L. Miller, and Todd Phillips. Sparse Voronoi Refinement. In *Proceedings of the 15th International Meshing Roundtable*, pages 339–356, 2006. Long version in Carnegie Mellon University Tech. Report CMU-CS-06-132.

[HPÜ05] 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.

[HT08] Benoît Hudson and Duru Türkoğlu. An efficient query structure for mesh refinement. In *Canadian Conference on Computational Geometry*, 2008.

[Hud07] Benoît Hudson. *Dynamic Mesh Refinement*. PhD thesis, School of Computer Science, Carnegie Mellon University, December 2007. Available as Technical Report CMU-CS-07-162.

[LT01] Xiang-Yang Li and Shang-Hua Teng. Generating well-shaped Delaunay meshes in 3D. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 28–37, 2001.

[LTU99] Xiang-Yang Li, Shang-Hua Teng, and Alper Üngör. Simultaneous refinement and coarsening for adaptive meshing. *Engineering with Computers*, 15(3):280–291, 1999.

[MBF04] Neil Molino, Zhaosheng Bao, and Ron Fedkiw. A virtual node algorithm for changing mesh topology during simulation. In *SIGGRAPH*, 2004.

[MTTW95] Gary L. Miller, Dafna Talmor, Shang-Hua Teng, and Noel Walkington. A Delaunay based numerical method for three dimensions: generation, formulation, and partition. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 683–692, 1995.

[Mul91] Ketan Mulmuley. Randomized multidimensional search trees (extended abstract): dynamic sampling. In *Proceedings of the seventh annual symposium on Computational geometry*, pages 121–131. ACM Press, 1991.

[NvdS04] Han-Wen Nienhuys and A. Frank van der Stappen. A Delaunay approach to interactive cutting in triangulated surfaces. In *Fifth International Workshop on Algorithmic Foundations of Robotics*, 2004.

[Rup95] Jim Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995.

[Sch91] Otfried Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 197–206, 1991.

[She97] Jonathan Richard Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3):305–363, 1997.

[STÜ07] Daniel Spielman, Shang-Hua Teng, and Alper Üngör. Parallel Delaunay refinement: Algorithms and analyses. *IJCGA*, 17:1–30, 2007.

[Tal97] Dafna Talmor. *Well-Spaced Points for Numerical Methods*. PhD thesis, Carnegie Mellon University, August 1997. Available as Technical Report CMU-CS-97-164.

A Quadtree

The balanced quadtree is originally defined by Bern, Eppstein, and Gilbert. We extend it to d dimensions. The quadtree is constructively defined as follows: we start with the root node $[0,1]^d$. We split quadtree nodes that are either *crowded* or *unbalanced*, largest first. A leaf node, i.e., a square, is *crowded* if it contains at least two vertices of N, or if it contains exactly one vertex of N, and one of its neighbors also contains a vertex. A square is *unbalanced* if one of its neighbors is one quarter its size, where the size of a square is the length of one of its edges. Upon splitting a node, we split it into 2^d children along its geometric middle. We keep neighbor pointers between the leaves of the quadtree: a square points to neighboring squares. Each square (not an internal node) stores a list of vertices that it contains; upon splitting, we reassign the vertices appropriately to the new children and change the original square into an internal node.

The $\texttt{QTAdd}(\Pi, v)$ routine starts a top-down search from the root of Π to find the square s that contains v, adds v to the list of vertices in s, then applies the recursive construction. The square s itself and its neighbor may now be crowded, which induces a set of splits. Crucially, we bound the splits geometrically:

Lemma A.1 *Consider a node s that is split due to crowding during $\texttt{QTAdd}(v)$. Then $|sv| \leq \sqrt{d}|s|$.*
Proof: Consider the state of the quadtree when s is split. We know that either s contains v, or s contains some vertex u and neighbors a square s' that contains v. In the former case, $|sv| = 0$. In the more interesting latter case, we know that QTAdd splits in order of size, largest first. So $|s| \geq |s'|$. At worst, s and s' meet at a corner and v is as far as possible: at the opposite corner of s'. Then $|sv| \leq \sqrt{d}|s'| \leq \sqrt{d}|s|$. ∎

Lemma A.2 *Consider a node s that is split due to balancing during $\texttt{QTAdd}(v)$. Then $|sv| \leq \sqrt{d}|s|$.*
Proof: By definition, when s is split due to balancing, it has a neighbor s' of one quarter its size. Before the call, s' cannot have existed, because then s would have been unbalanced. So we know that s' was created by QTAdd—in other words, its parent s'', of size $|s''| = |s|/2$, was split. It was split either due to crowding or balancing; inductively we can assume that $|s''v| \leq \sqrt{d}|s''|$. In the worst case, the line segment from v to s goes through s'' crossing the diagonal of s'': $|sv| \leq |s''v| + \sqrt{d}|s''| \leq \sqrt{d}|s|$. ∎

Lemma A.3 *QTAdd returns $O(\log \Delta)$ new squares and runs in $O(\log \Delta)$ time.*
Proof: Consider the square s returned by QTAdd that contains v. It has size at least $\ell = \delta'/4$, where δ' is the nearest neighbor distance of v in the new input. Consider all the squares returned by QTAdd that have size ℓ. They do not overlap, each has volume ℓ^d, and by previous lemmas, they all lie within distance $\sqrt{d}\ell$. A packing argument shows that there are $O(1)$ of them. The same holds for squares of size 2ℓ, 4ℓ, and geometrically up. In total, there are $O(\log \Delta)$ sizes of squares, thus, QTAdd returns $O(\log \Delta)$ new squares.

Furthermore, the work done amounts to finding the square that contains v, then splitting the squares. The former takes a search in a quadtree that has depth $O(\log \Delta)$. The latter takes only constant time to split each square, since the squares being split can only contain one vertex at most (excluding v), a total of an additional $O(\log \Delta)$ time. ∎

The QTBuild can be viewed as constructing a quadtree incrementally by calling QTAdd with every input vertex. Thus the bounds on QTBuild is n times those of QTAdd. The QTRemove routine does the reverse of QTAdd: it finds the square s that contains v, then checks whether any of the ancestors of s would have been split if v were not in the input, and also checks their neighbors. If so, processing ends. If not, it merges the siblings of s and/or of the neighbours', and repeats. Given that this exactly reverses the work that QTAdd does, all the same bounds hold, but with Δ referring to the input before QTRemove was called, rather than, as with QTAdd, referring to the input after QTAdd is invoked.

B Geometric Coloring

To bound the cost of a dynamic update, we find it critical to color the operations in the computation graph into a constant number of independent sets per rank. To achieve this, we first define the notion of independence. Then, we show that we can compute a coloring using just a simple geometric calculation (due to Spielman, Teng, and Üngör [STÜ07]). At rank r, the static algorithm executes all the dispatch operations before executing any fill operation, and dispatch operations do not change the set of vertices. Therefore, execution of the dispatch operations at rank r does not depend on the order we execute them. For fill operations, this is not the case. A fill operation may insert a Steiner vertex which may affect the β-clipped Voronoi cell computation of another fill operation, provided that the vertices these operations act on are sufficiently close to each other. More specifically, we say that two fill operations at rank r acting on vertices v_1 and v_2 are *independent* if the function calls $\texttt{QTClippedVoronoi}(v_1, \beta)$ and $\texttt{QTClippedVoronoi}(v_2, \beta)$ cannot possibly read the same quadtree square.

Recall that $\texttt{QTClippedVoronoi}(v, \beta)$ only accesses squares s that are close to v: for all $x \in s$, $|vx| \in O(\mathrm{NN}_{\mathcal{M}}(v))$. Let $\gamma_0 > 1$ be the constant in the big-Oh notation. For fill operations at rank r, Lemma C.1 and the fact that our algorithm does not schedule fill operations for a rank smaller than the current one guarantee that the nearest neighbors of v_1 and v_2 are at most ρ^{r+1} away. Using $\gamma = \rho\gamma_0$, we see that two fill operations at rank r acting on vertices v_1 and v_2 are independent if $|v_1v_2| < 2\gamma\rho^r$. To exploit this observation, we impose a grid upon space — a periodic square lattice — such that every unit square of the lattice contains at most one active vertex. We guarantee this by simply making sure that the farthest points in a unit square are at most $2\gamma\rho^r$ far apart. The scale of the lattice, i.e., the length of one of the edges of the unit square, ℓ_r, should satisfy $\ell_r\sqrt{d} < 2\gamma\rho^r$; we choose $\ell_r = \rho^r/\sqrt{d}$.

The period of this lattice, κ, is the number of colors we use for every dimension. The color of a fill operation *op* at rank r acting on vertex v can be calculated as follows: take each coordinate of v, divide by ℓ_r, take the floor, and take modulo κ. The coordinates of v transformed in this way is the color of *op*. Note that there are κ^d possible values of colors used for coloring a fill operation. For independence among fill operations of the same color, it is sufficient to ensure that same colored lattice squares are at least $2\gamma\rho^r = 2\gamma\ell_r\sqrt{d}$ far apart. Using κ colors, this distance is $(\kappa - 1)\ell_r$; choosing $\kappa > 1 + 2\gamma\sqrt{d}$ grants independence between the fill operations of the same color.

C Output Quality and Size

Lemma 4.1 *For all t, M_t is size-conforming. Consequently, M is size-optimal.*

Proof: By definition of size-conforming we want to show that for every $v \in \mathsf{M}_t$, we have $\mathrm{NN}_t(v) \in \Omega(\mathrm{lfs}(v))$. This proof is inductive over the order in which the algorithm inserts the vertices. In the base case, every vertex is an input vertex and the nearest neighbor of a vertex v is exactly the local feature size. For the inductive case, assume that for every vertex $v \in \mathcal{M}$ we have $c\,\mathrm{NN}_{\mathcal{M}}(v) \geq \mathrm{lfs}(v)$, where c is the constant in the asymptotic notation. Furthermore, assume that v inserts a Steiner vertex u and the new output is $\mathcal{M}' = \mathcal{M} \cup \{u\}$. We analyze the inductive claim for u and for any vertex $w \in \mathcal{M}$ separately. For u we know that $\mathrm{NN}_{\mathcal{M}'}(u) = |uv|$ and that u is chosen far from v: $|uv|/\rho \geq \mathrm{NN}_{\mathcal{M}}(v)$. It is known that lfs satisfies the Lipschitz condition: $\mathrm{lfs}(v) + |uv| \geq \mathrm{lfs}(u)$. By the inductive hypothesis, $c\,\mathrm{NN}_{\mathcal{M}}(v) \geq \mathrm{lfs}(v)$. Therefore, we have $\left(\frac{c}{\rho} + 1\right)|uv| \geq \mathrm{lfs}(u)$. Now, for any vertex $w \in \mathcal{M}$, if $\mathrm{NN}_{\mathcal{M}}(w) = \mathrm{NN}_{\mathcal{M}'}(w)$ then the claim holds trivially. So, assume $\mathrm{NN}_{\mathcal{M}}(w) > \mathrm{NN}_{\mathcal{M}'}(w) = |uw|$. By the Lipschitz condition, $|uw| + \mathrm{lfs}(u) \geq \mathrm{lfs}(w)$. Since u is in the Voronoi cell of v, $|uw| \geq |uv|$. Combining this fact by the bound we obtained for $\mathrm{lfs}(u)$,

we get $\left(\frac{c}{\rho}+2\right)|uw| \geq \mathrm{lfs}(w)$. Solving for $c \geq \left(\frac{c}{\rho}+2\right)$, we conclude that $c \geq \frac{2\rho}{\rho-1}$ suffices. ■

The Progress Lemma appeals to the claim that every vertex has a fill operation scheduled at the "right" time. We make this precise and prove the claim.

Lemma C.1 *At time $t=(r,0)$, assume that every vertex $u \in \mathsf{M}_t$ with $\mathrm{NN}_t(u) < \rho^r$ is ρ-well-spaced. Then, for every vertex $v \in \mathsf{M}_t$ with $\mathrm{NN}_t(v) \in [\rho^r, \rho^{r+1})$, there exists a fill operation in Ω that acts on v at rank r.*

Proof: Let v be a vertex satisfying the premises and u be the nearest neighbor of v at time $t=(r,0)$. If we can prove that the dispatch operation that acts on v runs at rank $\leq r$ and u is a nearest neighbor at that time then this operation schedules a fill operation at rank r. Alternatively, it is also enough to prove that the dispatch operation that acts on u runs at rank $r' \leq r$ and v is a β-clipped Voronoi neighbor of u at time $t'=(r',0)$. Analyzing the cases: if both v and u are input vertices then the dispatch operation of v is scheduled to run at rank $\leq r$. In the case that v is a Steiner vertex and u already exists when v is being created, consider the vertex v' that creates v. Since v is in the Voronoi cell of v' at the time of creation, we have $|vv'| \leq |uv|$, which implies that the dispatch operation that acts on v must be running at rank $\leq r$. The final case to consider is that u is a Steiner vertex and v already exists at the time of creation of u. Similar to the previous case, we can say that the dispatch operation that acts on u must be running at rank $r' \leq r$. Since u is the nearest neighbor of v at time t, v is a Voronoi neighbor of u at time t'. If u is already ρ-well-spaced at time t then $|vu| \leq 2\rho\,\mathrm{NN}_t(u) < 2\beta\,\mathrm{NN}_{t'}(u)$. Otherwise, by the assumption of the lemma, we deduct $\rho^r \leq \mathrm{NN}_t(u)$, which again implies $|vu| = \mathrm{NN}_t(v) < \rho\,\mathrm{NN}_t(u) < 2\beta\,\mathrm{NN}_{t'}(u)$. Either way, v is a β-clipped Voronoi neighbor of u at time t'. ■

D Runtime

Lemma 5.1 *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 other direction, we analyze Steiner vertices and input vertices separately. An input vertex v by definition has $\mathrm{lfs}(v) = \mathrm{NN}_0(v)$. The algorithm is size-conforming (Lemma 4.1), so $\mathrm{NN}_0(v) = \mathrm{lfs}(v) \in O(\mathrm{NN}_\infty(v))$. When a Steiner vertex v is created at time $t=(r,c)$, its nearest neighbor is at distance at most $\beta\rho^{r+1}$ and at least ρ^{r+1}. Any other vertex u created during the processing of rank r or later also has its nearest neighbor upon creation at distance at least ρ^{r+1}; in particular, $|uv| \geq \rho^{r+1}$. Thus $\mathrm{NN}_t(v) \leq \beta\rho^{r+1} \leq \beta\,\mathrm{NN}_\infty(v)$. ■

Lemma 5.2 *If an operation acts on v at rank r, then $\mathrm{NN}_\infty(v) = \Theta(\rho^r)$.*

Proof: First we prove the upper bound. Consider a dispatch operation that acts on an input vertex v. The rank of this operation is calculated as $r = \lfloor \log_\rho \mathtt{QTApxNN}(v) \rfloor$, which implies $r \geq \lfloor \log_\rho (1-\epsilon)\,\mathrm{NN}_0(v) \rfloor$. In other words, $\mathrm{NN}_\infty(v) \leq \mathrm{NN}_0(v) \leq \rho^{r+1}/(1-\epsilon)$. Out of the dispatch operations of Steiner vertices and the insert-Steiners operations of all the vertices, assume an operation *op* that acts on v is scheduled at rank r or at time t. Then we know that $r = \lfloor \log_\rho |uv| \rfloor$ for a vertex u (an operation that acts on u schedules *op*). Since $\mathrm{NN}_t(v) \leq |uv|$, we get $\mathrm{NN}_t(v) < \rho^{r+1}$. Therefore, the upper bound holds: $\mathrm{NN}_\infty(v) \leq \mathrm{NN}_t(v) = O(\rho^r)$.

For the lower bound, consider an operation *op* that acts on v at rank r and assume that *op* was scheduled at time t. Trivially, $\mathrm{lfs}(v) \geq \mathrm{NN}_t(v)$ and by Lemma 4.1, we know that $\mathrm{NN}_\infty(v) = \Omega(\mathrm{lfs}(v))$; therefore it

suffices to show that $\mathrm{NN}_t(v) = \Omega(\rho^r)$. If v is an input vertex and *op* is a dispatch operation, then by the guarantees of the call QTApxNN (v), we get $\mathrm{NN}_0(v) = \Omega(\rho^r)$ and the proof is done. Otherwise, assume that the operation that schedules *op* acts on u at rank $r' < r$ or at time t. Since this operation schedules *op* for rank r, $r = \lfloor \log_\rho |vu| \rfloor$, which implies $|vu| \geq \rho^r$. If $\mathrm{NN}_t(v) < \rho^{r'}$, then by Progress Lemma, v is ρ-well-spaced at time t. Furthermore, we know that v is a β-clipped Voronoi neighbor of u, which implies that u is a ρ-clipped Voronoi neighbor of v at time t. Therefore, $2\rho\, \mathrm{NN}_t(v) \geq |vu| \geq \rho^r$ and we are done. In the other case, $\mathrm{NN}_t(v) \geq \rho^{r'}$. Again, since v is a β-clipped Voronoi neighbor of u, $|vu| \leq 2\beta\, \mathrm{NN}_t(u)$. Applying the upper bound result from the first part for the operation that acts on u, we get $\mathrm{NN}_t(u) = O(\rho^{r'})$; thus $|vu| = O(\rho^{r'})$. This implies $\mathrm{NN}_t(v) = \Omega(|vu|)$ and $\mathrm{NN}_t(v) = \Omega(\rho^r)$ as desired. ∎

Lemma 5.4 *For every vertex $v \in \mathsf{M}$, there are $O(1)$ operations that act on v.*

Proof: By Lemma 5.2, 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 are reduced to counting fill operations scheduled by other dispatch operations. Now, 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 must be in $\mathrm{Vor}^\beta_{t'}(u)$, consequently $|uv| \leq 2\beta\, \mathrm{NN}_{t'}(u)$. The fact that the fill operation is scheduled for rank r implies $\rho^r \leq |uv| < \rho^{r+1}$. Considering the dispatch operation, Lemmas 5.1 and 5.2 show that $\mathrm{NN}_{t'}(u) = O(\rho^{r'})$. These facts altogether imply $\rho^r = O(\rho^{r'})$. Again by Lemma 5.2, 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 suffices to prove our claim. ∎

E Dynamic

Consider two executions of the static algorithm with inputs N and N' that are related by the insertion/deletion of some point v^*. Let M be the output and $G = (\Omega \cup \Sigma, E)$ be the computation graph of the execution with N and let M' be the output and $G' = (\Omega' \cup \Sigma', E')$ to be the computation graph of the execution with N'.

Lemma 6.1 *For every operation in $\Omega^- \cup \Omega^+ \cup \Omega^\times$, there exists a dependence path from a square in $\Sigma^\times$.*

Proof: By definition of inconsistent operations, an operation $op \in \Omega^\times$ can be reachable via a dependence path from $\Sigma^\times$. For unmatched operations, i.e., operations in $\Omega^- \cup \Omega^+$, assume towards a contradiction that there exist some that cannot be reachable. Let *op* be the earliest of such operations. Since *op* does not depend on an inconsistent square, it does not read an inconsistent square. Let us assume that *op* is a dispatch operation acting on an input vertex v, which must be a common vertex to both executions. Thus, v lies in identical squares of two executions, which implies that QTApxNN returns the same rank for v in both executions. Then, the definition of μ_o covers *op* since there exists an operation *op*′ in the other execution which is identical to *op*: *op* and *op*′ act on v at the same time. 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 dependence path from a square in $\Sigma^\times$. Extending that dependence path to *op* proves the contradiction. ∎

Lemma 7.1 *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 $B = \Omega^{-} \cup \Omega^{+} \cup \Omega^{\times}$. We prove the equality by showing containment in both directions. Towards a contradiction, assume that $A \not\subset B$. 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*, $op' \in B$, 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 $\texttt{MarkReaders}(s, t)$ with a smaller time t than the time of *op*. If $s \in \Sigma^{\otimes}$ then clearly $op \in B$; if not, there must be another operation op' running at time t that writes into s. Again, by minimality of *op*, $op' \in B$ and there exists a dependence path from op' to *op* which puts *op* in B. Contradiction.

For the other direction, similarly assume the contrary 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 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 dependence path from a square $s \in \Sigma^{\times}$ to *op*. Pick the longest dependence 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 `Propagate` puts *op* in A. In the other case that op' exists, by minimality of *op*, op' is in A and the dependence 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 dependence between *op* and op'. Contradiction. ■

When the update algorithm is executed, it transforms the first execution to the second one by updating the computation graph (G to $\tilde{G}$), consequently the output (M to $\tilde{\mathsf{M}}$).

Lemma 7.2 (Isomorphism) *The output sets $\tilde{\mathsf{M}}$ and M' are equal and there is an isomorphism $\phi : \tilde{G} \rightarrow G'$ that preserves the vertex and time of each operation.*

Proof: We prove equality of the output and build ϕ inductively. Define the sets of operations according to their creation times: $\Omega_t^{\ominus} = \{op \in \Omega^{\ominus} \mid op \text{ is created at time } < t\}$. For $t = 0$ we say $\Omega_0^{\ominus} = \{\texttt{Dispatch} \text{ of } v \mid v \in \mathsf{M}_0^{-}\}$. We 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 $\geq t$; these are the edges related to the execution of the operations in $\tilde{\Omega}_t$ scheduled to execute at time $\geq t$. Similarly, let G'_t be the subgraph of G' induced by the nodes $\Omega'_t \cup \Sigma'$ excluding the edges with time $\geq t$.

Initially, $\tilde{\mathsf{M}}_0 = \mathsf{M}'_0 = \mathsf{N}'$ and $\tilde{\Sigma} = \Sigma'$. Therefore, there exists a natural correspondence between the dispatch operations in $\tilde{\Omega}_0$ and Ω'_0 and we have a natural isomorphism $\phi_0 : \tilde{G}_0 \rightarrow G'_0$. For the inductive step at time t, assume that $\tilde{\mathsf{M}}_t = \mathsf{M}'_t$ and that we have an isomorphism $\phi_t : \tilde{G}_t \rightarrow G'_t$ which preserves the vertex and the time of each operation. Pick any node $op \in \tilde{\Omega}_t$ that is scheduled to execute at time t. 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 $\Omega_t^{\oplus}$, in $\Omega_t^{\otimes}$, or it is not scheduled in any of the $\Omega^{\ominus}$, $\Omega^{\oplus}$, and $\Omega^{\otimes}$ sets, i.e., it is consistent.

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 M_t^{-} because the vertices in M_t^{-} 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 a consistent operation. 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 is not in $\Sigma^\times$, i.e., s does not contain a vertex in $\mathsf{M}_t^- \cup \mathsf{M}_t^+$. 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.

Then, we have a natural correspondence between the operations that op and op' create. If op and op' are fill operations, then the Steiner vertices they create are at the same coordinates. Therefore, $\tilde{\mathsf{M}}_{t+1} = \mathsf{M}'_{t+1}$. Furthermore, because op and op' read and write the same squares in $\tilde{\Sigma} = \Sigma'$, the edges between these operations and the squares have natural correspondences as well. Extending ϕ_t to ϕ_{t+1} by adding these correspondences completes proof of the inductive step. ■

F Lower Bound

We define a gadget (see Figure 5) consisting of points in the hypercube $[0, k^{-1/d}]^d$. We have two vertices at distance δ from each other in the middle of the box; we say that one of them is the "dynamic" vertex that will be inserted. We also have a grid of $O(1)$ vertices on each of the faces of the hypercube, chosen according to the scheme of Hudson [Hud07, p.79]. Our initial input consists of tiling $[0, 1]^d$ with the gadgets, without any dynamic vertex. Our insertion sequence consists of inserting the k dynamic vertices, one for each gadget.

Lemma F.1 *Inserting the dynamic vertex to a single gadget requires adding $\Omega(\log \Delta)$ Steiner vertices.*

Proof: Let N be the set of vertices before adding the dynamic vertex v; also, let $\mathsf{N}' = \mathsf{N} \cup \{v\}$. Draw the segment from v to the farthest point in $\mathrm{Vor}_{\mathsf{N}'}(v)$. This segment has length at least $\ell = \frac{1}{4} - \frac{\delta}{2}$. Consider the Voronoi diagram after computing a ρ-well-spaced superset of N' and consider the Voronoi cells that this segment cuts. Let $q_1 \dots q_k$ be the vertices of those Voronoi cells, in order. We know that the Voronoi cells in the output have aspect ratio $< \rho$, therefore, $|q_1 v| \leq 2\rho \, \mathrm{NN}_{\mathsf{N}'}(v) = 2\rho\delta$. So the nearest neighbour distance of q_1 is at most $|q_1 v|$. We can use the same argument to get $|q_1 q_2| \leq 2\rho |q_1 v|$ and repeat. In other words, distance from v grows only geometrically as we walk down the segment: covering the distance ℓ requires $\Omega(\log 1/\delta) = \Omega(\log \Delta)$ many Steiner vertices to be inserted. ■

Lemma F.2 *Inserting k dynamic vertices to an initial input of k gadgets requires inserting $\Omega(k \log \Delta)$ Steiner vertices.*

Proof: We refer to a technique of inserting vertices to the hypercube faces [Hud07]. 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 well-spaced superset must contain at least $\Omega(k \log \Delta)$ Steiner vertices, for a carefully selected ρ. Since, there exists a constant $\rho > 1$ such that the original input of k gadgets is ρ-well-spaced, our proof is complete. ■