Dynamic Mesh Refinement with Quad Trees and Off-Centers

Umut A. Acar[†] **Benoît Hudson**

April 20, 2007
CMU-CS-07-121

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Many algorithms exist for producing quality meshes when the input point cloud is known *a priori*. However, modern finite element simulations and graphics applications need to change the input set during the simulation dynamically. In this paper, we show a dynamic algorithm for building and maintaining a quadtree under insertions into and deletions from an input point set in any fixed dimension. This algorithm runs in $O(\lg L/s)$ time per update, where L/s is the spread of the input. The result of the dynamic quadtree can be combined with a postprocessing step to generate and maintain a simplicial mesh under dynamic changes in the same asymptotic runtime. The mesh output by the dynamic algorithm is of good quality (it has no small dihedral angle), and is optimal in size. This gives the first time-optimal dynamic algorithm that outputs good quality meshes in any dimension. As a second result, we dynamize the quadtree postprocessing technique of Har-Peled and Üngör for generating meshes in two dimensions. When composed with the dynamic quadtree algorithm, the resulting algorithm yields quality meshes that are the smallest known in practice, while guaranteeing the same asymptotic optimality guarantees.

[†] Toyota Technological Institute at Chicago

This work was supported in part by the National Science Foundation under grants ACI 0086093, CCR-0085982 and CCR-0122581.

Keywords: Computational Geometry, Mesh Refinement, Dynamic Algorithms, Self-Adjusting Computation

1 Introduction

In many applications, we need to *mesh* or a *triangulate* a domain consisting of points and features by splitting it into triangles such that all elements of the domain are covered by a union of triangles. Meshes are typically used to interpolate a continuous function for any of various purposes such as finite element simulations or graphics. A subtantial amount of research has been performed on the *static meshing problem* [Che89, BEG90, MV92, Rup95, She98, ...] which assumes that the input domain is known a priori. We are interested in the *dynamic meshing problem* which permits the input to be changed. For the purpose of this paper, we assume that the input consists of points and that the input can be changed by inserting new points and deleting existing points.

To be broadly applicable, a dynamic meshing algorithm must satisfy the properties satisifed by state-of-the art *static* meshing algorithms; the dynamic setting in turn imposes additional requirements. These properties concern the relationship between input, and the output, the quality of the output, and work efficiency. First the algorithm must yield *conforming* meshes, *i.e.*, all points in the input must appear as a corner of a triangle in the ouput. Second, the output must be *good quality*, *i.e.*, the internal angles of the triangles in the output must be bounded away from 180 °. Third, the ouput must be *size competitive*, *i.e.*, the number of triangles in the output must be as small as possible. Fourth, the algorithm must be *(work) efficient*, *i.e.*, it should preprocess the input quickly. Fifth, the algorithm must be *responsive*, *i.e.*, it should respond to insertions and deletions by updating its output quickly. Finally, it is often desired that the output of a dynamic meshing algorithm be *history independent*, *i.e.*, the output mesh is equal to a mesh of the current input set and does not depend on the history of the operations (insertion and deletions performed).

These properties can be broadly placed in two categories: those required by the application (conformity, good quality output, size optimality, and history independence) and those that concern performance (work efficiency and size optimiality). One major application for meshing is in the Finite Element Method (FEM) of scientific computing [Joh87, for example]. In FEM simulations, mesh quality is important because it determines the simulation error [BA76]; the number of elements in the mesh is important because it determines the simulation runtime; the size of the smallest element is important because it defines the length of the timestep. Furthermore, some applications [HPÜ05] require that the size of mesh elements be locally determined by the properties of the input such as the local feature size; it is then important that the output be faithful to the input. The last two concerns motivate a need for history independence: that the output depend only on the current set of points and not the sequence of operations performed to obtain that input set.

The meshing problem has been studied extensively since 1950s. The first meshing algorithms that could generate provably good quality meshes using only a constant factor more elements (e.g., simplices) than optimal only emerged in the early 1990s, with work from Bern, Eppstein, and Gilbert [BEG90]. The Bern-Eppstein-Gilbert algorithm was later extended to three and higher dimensions by Mitchell and Vavasis [MV92, MV00]. Both these solutions run in time $O(n \lg n + m)$, where n is the number of input points, and m is the number of output elements in the optimal result. The technique uses a *quadtree subdivision*, and warps the vertices of the mesh in a postprocess, triangulating in a final step. A number of postprocesses have been conceived, the most recent of which produces meshes smaller than any other published technique [HPÜ05]. The time bound is optimal, due to a sorting lower bound. The size of the mesh they output is also provably within a constant factor of optimal.

Our results. To state our results, we start with a few definitions for characterizing the input. As usual, n is the number of points in the input (in the dynamic case, it is the number of points in the current input, irrespective of history and future). We assume that the smallest possible mesh has exactly m vertices in the output. L is the diameter of the point set, and s is the distance between the closest pair, so that L/s is the *spread* of the input. Finally, d is the dimensionality of space. We consider d to be a constant, which allows us to hide terms exponential in the dimension from the asymptotics; none of the results mentioned in this paper or its references directly apply to high-dimensional problems. Generally, d will be 2 or 3; however, our results apply in any fixed dimension.

We give a mesh refinement algorithm that runs in $O(n \lg L/s)$ time to preprocess a static point set with n points. The preprocessing step yields a quality mesh, as described above. After the preprocessing step, the input can be changed by inserting new points and deleting existing points. To each such change, the algorithm responds in $O(\lg L/s)$ time by updating the output mesh. The algorithm is history independent with respect to the preprocessing-step: the output mesh is identical to the mesh that would have been obtained by performing a preprocessing step from scratch. The algorithm thus guarantees that the output retains all its guarantees regardless of the operation sequence.

The response time of $O(\lg L/s)$ is optimal in two senses: first, the output mesh may, in the worst case, change by $O(\lg L/s)$. Second, under the assumption that the input has polynomial spread, the response time is $O(\lg n)$, matching the lower bound based on sorting. We believe that the proposed algorithm is the first optimal-time dynamic mesh refinement algorithm with output quality and size guarantees.

To solve the dynamic meshing problem, we first show how to dynamize the construction of a so-called *balanced quad-tree* [BEG90]. Having produced a dynamic balanced quad-tree algorithm, we show how to dynamize a postprocess to produce a quality simplicial mesh. The original post-processes described in the early quad-tree papers produce meshes that, in practice, are larger than subsequently discovered mesh refinement approaches based on Delaunay refinement [She98, Rup95]. For two dimensions, a recent result of Har-Peled and Üngör [HPÜ05] describes a more complicated post-process that achieves the smallest meshes known—Shewchuk has integrated Üngör's technique in recent versions of his well-known Triangle software and finds it reduces the mesh size by another 40% over the usual approach [Üng04]. Our history-independent dynamization simulates this algorithm exactly, and thus will achieve the same output size.

In order to dynamize the algorithms, we make very minor modifications to them – we specify the order of some operations that the original authors left arbitrary – then apply *self-adjusting computation* [Aca05a]. In self-adjusting computation, after a static algorithm is executed with some input, any of the computation data can be changed and the output can be updated by running a *change propagation* algorithm. At a high-level, the change-propagation algorithm updates the output as though the algorithm is executed from-scratch on the changed input, but only re-executes the parts of the computation that depend on the changed data. Previously the approach has been applied to a reasonably broad range of problems for both dynamizing and kinetizing various algorithms [ABT07, ABTV06, ABBT06]. Self-adjusting-computation yields a dynamic algorithm that guarantees that dynamized algorithm is correct, history-independent, and composable. Composibility and history-independence are critical to combining our dynamic quadtree algorithm with the postprocessing technique for Har-Peled and Üngör: without these the bounds would break. For programs that satisfy a so-called *monotonicity* condition, the time for dynamic changes depends on the *trace stability* of the algorithm. To obtain our results, we show that both the quad-tree algorithm and the postprocesses satisfy the monotonicity conditions, and we bound their trace stability.

Related work. A number of authors have considered the dynamic mesh refinement problem [NvdS04, MBF04, CGS06, and references, for example], especially for surgical simulation applications (where a scalpel cut introduces new features) and in fracture simulations. Published solutions either either do not guarantee mesh size, or do not guarantee quality. None of them guarantee runtime, though for some solutions it is, at least in practice, faster than linear time to change the topology of the mesh. The related *moving mesh*

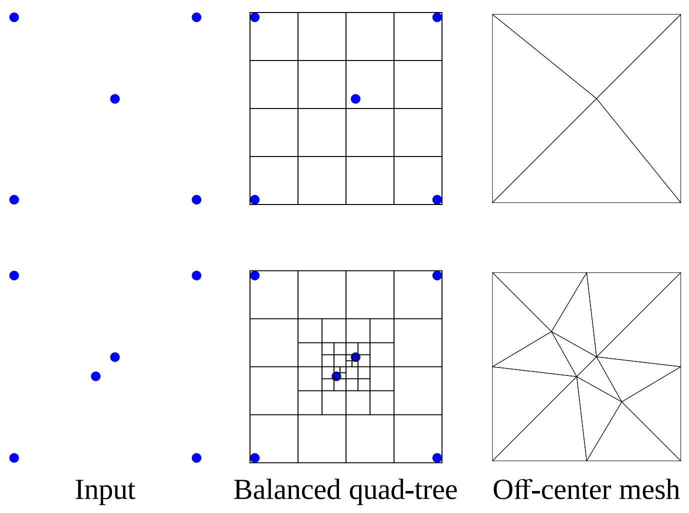

Figure 1: An illustration of our dynamized quadtree algorithm, showing the differences between two static runs. Under self-adjusting computation, the total work done to update from one input to the other is linear in the change between the two. Section 4 bounds the change in the balanced quad-tree, whereas Section 5 bounds the change in the output mesh. Both are in $O(\lg L/s)$, where L is the diameter of the space and s is the distance between the nearest pair of input points.

problem, where all the points in the mesh move through time, has been attacked both in practice [Bak01, e.g.] and theoretically [LTÜ98, e.g.] in a huge number of works, mostly with application to computational fluid dynamics. However, all current moving mesh approaches take at least linear time.

We apply self-adjusting computation [Aca05a] to dynamize a static algorithm. Our algorithms could be dynamized using other techniques instead. For example, deletions can be handled lazily by delaying the removal of the deleted point until a sufficiently large (near-linear) number of points are deleted, and then remeshing from scratch. Such an algorithm can be made to be size-optimal and, in an amortized sense, has near-optimal response time. However, it is of course not history independent. Even if this is appropriate for the reader's application (it is not for those we have in mind), the reader will be interested in our analysis to bound the time to perform a dynamic insertion. Another class of dynamization techniques include those for order-decomposable search problems [Ove81]. This approach, however, only applies to divide-and-conquer algorithms. It is not clear how to restate the algorithms we dynamize here in a divide-and-conquer framework: splitting a cell can cause splitting another cell that is arbitrarily far away, which seems to contravene any division possibilities.

Eppstein *et al.* reported on the dynamization of a quad-tree [EGS05] using skip quadtrees. To establish their fast query and update times, they must cleverly compress away uninteresting quad-tree cells and maintain a hierarchy of trees. However, even if we used skip quadtrees as our backing data structure, we would still need the analysis in this paper to show we can quickly maintain the balance condition on the quadtree, and to show how to update the output mesh: dynamizing the underlying data structures is not the hard part of dynamic mesh refinement.

```
QuadTreeRefine($\mathcal{P}$: point set, $L$: real, $d$: int)
1   Associate $\mathcal{P}$ with the cell $[0, L]^d$
2   If $[0, L]^d$ is crowded then { addWork($[0, L]^d$) }
3   $l \leftarrow \lg L$
4   while ($|W| > 0$) do
5       while ($|W_l| = 0$) do { decrement $l$ }
6       splitAll($W_l$)
7       increment $l$

split($c$: cell)
8   Split $c$ into $2^d$ new, smaller cells $\{c_i\}$
9   for (each point $p$ contained by $c$) do
10      associate $p$ with the $c_i$ that contains it
11  return $\{c_i\}$
```

```
addWork($c$: cell)
1   Append $c$ to $W_{\lg|c|}$

splitAll($W_l$: cell set)
2   newcells $\leftarrow \emptyset$
3   while ($W_l$ not empty) do
4       dequeue $c$ from $W_l$
5       $\{c_i\} \leftarrow$ split($c$)
6       append each $c_i$ to newcells
7   while (newcells not empty) do
8       dequeue $c_i$ from newcells
9       if ($c_i$ is crowded) then { addWork($c_i$) }
10      for (each neighbour $c_i'$ of $c_i$) do
11          if ($|c_i'| \geq 4|c_i|$) then { addWork($c_i'$) }
```

Figure 2: The quadtree refinement algorithm, modified from Bern, Eppstein, and Gilbert [BEG90].

2 Balanced Quadtrees

Balanced quadtrees yield a hierachical subdivision of the space into *cells*, i.e., hypercubes in the specified dimension. Having created a balanced quadtree of a set of inputs, a post-processing step can be applied to produce a good-quality mesh of the input. In this section, we present a modification of the original algorithm of Bern, Eppstein, and Gilbert [BEG90].

A cell is a hypercube in the specified dimension. Importing the definitions of Bern *et al.*, we say that a cell c is *self-crowded* if it contains two or more input point. A cell c is *crowded by a neighbour* c' if c contains exactly one point, and c' contains are least one point. We say that a cell is *crowded* if it is self-crowded or is crowded by a neighbor. We say that a cell c is *unbalanced* if it has a neighbour c' such that $|c|/|c'| \geq 4$. We say that a quadtree is *balanced* if all unsplit cell are balanced and are not self-crowded or crowded.

Figure 2 shows our quadtree algorithm. In essence, our algorithm is a restatement of Bern *et al.* in which we more carefully specify the ordering of some operations that was left undefined in the original work. The ordering was unimportant to their setting, but it is critical for our proof of the response time bounds. The algorithm starts with a bounding box (square) of the the point set, with side length L. It maintains a set W of work items, i.e., cells to split, and a mapping from each cell to the set of input points that it contains. The work-set W is partitioned into $\lg L$ buckets such that the bucket W_i is a queue containing the cells of size exactly 2^i. The main loop maintains a finger l in order to quickly find the largest non-empty bucket.

The algorithm proceeds in rounds. In each round, it chooses the set of the largest cells on the workset and splits all of them using the splitAll function. The splitAll first splits each cell in the bucked by calling split. The split function splits the cell into 2^d sub-cells and updates the cell-to-points mapping. The splitAll function then enqueues the newly-created crowded or unbalanced cells into the work set by calling addWork, which is only a function in order for us to easily refer to it throughout the paper. The function addWork is the only operation that causes communication across cells: indeed, only an unbalanced cell can be added more that once to the workset (by different neighbours). We discuss later the implications of this operation. At the end of one round of split operations performed by splitAll the algorithm increments the main loop's finger, since splitAll may have unbalanced some cells that are larger than the cells previously being processed.

2.1 Structural Results

Lemma 2.1 *During the algorithm, unprocessed crowded cells (if any exist) are all of the size of the smallest cells in the mesh.*

Proof Initially, this is trivially true (there is only one cell in the mesh). Later, consider the cell c^+ that was split to create a crowded cell c. Clearly, c^+ was itself crowded, and thus by induction was the smallest cell in the mesh. Now, we have destroyed c^+ and all its equally-sized cells, and replaced them with cells of half the size. These new cells must be the smallest cells in the mesh. Until we split these crowded cells, any further splits must all be balance splits. A cell can only be unbalanced if it is four times larger than its neighbour, thus balance splits cannot reduce the size of the smallest cell. ■

Lemma 2.2 *After a round of splitting crowded cells, until the next round of splitting crowded cells, l increases by exactly one every round.*

Proof When splitting the crowded cells, we know that all cells in the mesh are balanced: there are no smaller cells, and any larger cells, if unbalanced, would imply a work set $W_{l'}$ with $l' > l$ was non-empty, a contradiction. The crowded cells may cause unbalanced cells, with size corresponding to $l + 1$, but not of size $l + 2$ because such cells would already be unbalanced, a contradiction. ■

Lemma 2.3 *At all points in the algorithm, every cell c has at most $O(1)$ neighbours c' of size $4|c'| \leq |c| \leq 0.25|c'|$.*

Proof The proof that the size does not differ much is immediate from the prior lemma. The proof that this implies a bounded number of neighbours is by a volume packing argument. The constant is precisely $6^d - 4^d$. ■

2.2 Size and quality guarantees

To obtain the size and quality guarantees, we can use any of the standard postprocesses published in Bern *et al.* or Mitchell and Vavasis [BEG90, MV00]. Given that our algorithm is just a specific ordering consistent with the schema given by the prior results, we inherit the size and quality guarantees. For example, we can show that all the simplices have aspect ratio at least some constant that depends only on the dimension, and not on the input point set. Furthermore, we can show that among all Steiner triangulations that respect that aspect ratio bound and in which all the input points appear, the size of the triangulation output by the quadtree algorithm and its postprocess is within a constant factor of optimal. In fact, the bound is stronger: at any point p in the domain, we know that the cell that contains p has size within a constant factor of the local feature size at p (the distance from p to the second-nearest input point).

2.3 Blame argument

Definition 2.4 *If a new cell c_i is crowded by a point p in c_i or in a neighbour of c_i, then we* ***blame*** *the split of c_i on p. Inductively, if a split of a cell c' blamed on p causes a cell c'' to become unbalanced, we blame the split of c'' on p.*

Note that a cell may blame its splitting on many points; indeed, it will always blame at least two points.

Lemma 2.5 *Assume p is blamed for the split of a cell c. Then $\|pc\| \in O(|c|)$.*

Proof If c is being split for crowding, then p is either within c or is in a neighbour c' of c, and $|c'| = |c|$. Thus $||pc|| \le |c|$.

If c is being split for balance, then we can follow the causal chain that leads to a cell c' that was split for crowding by p. Label the chain c_i with $c_0 = c$ and $c_k = c'$. Because of the balance condition, we know that $|c_i| = 2|c_{i+1}|$ and thus $|c| = 2^k|c'|$.

The distance we can travel along the chain is maximized if the chain follows the diagonal of the cells, a total distance of $2^k\sqrt{2}|c'|$. Finally, c' either contains p or neighbours an equal-sized cell that contains p. Thus the distance from p to c is at most $(2^k\sqrt{2} + 1)|c'|$.

In other words, $||pc|| < (\sqrt{2} + 1)|c|$. ■

Lemma 2.6 *Any point p is blamed for at most $O(\lg L/s)$ splits.*

Proof Given a size class l, we know that any cell of size 2^l that is blamed on p must have distance at most $O(2^l)$. A simple packing argument shows that there must thus be only $O(1)$ splits in size class l that are blamed on p. Because the algorithm does not overrefine, there are $O(\lg L/s)$ size classes. ■

2.4 Runtime

There are two components to the runtime of the algorithm: the cost of splitting the cells, and the cost of maintaining the mapping between points and cells. Each point is blamed for $O(\lg L/s)$ splits, so there are a total of at most $O(n \lg L/s)$ splits. If a split relocates a point, there are two possibilities: the split is due to crowding, in which case the point is blamed for the split; or the split is due to balance, in which case there is at most one point in the cell. In the former case, the cost of the relocation can be charged to the point in the usual manner. In the latter case, the cost can be charged to the split itself since it is only constant extra work.

3 Self-Adjusting Computation

The *self-adjusting computation* (SAC) model [Aca05a] enables dynamizing static algorithms automatically by relying on a *change-propagation algorithm* to update the output when the input changes. The asymptotic complexity of change propagation can be bound by analyzing the *trace stability* of the algorithm under a change—in this paper , we consider inserting or deleting one point from the input. In this section, we state some definitions that our analysis (Section 4.2) relies on. For brevity and to draw on the reader's intuition, we paraphrase from the more precise definitions in Acar's presentation [Aca05a] and present the main stability or update theorem that change propagation time can be bound by stability and a priority-queue overhead for certain programs.

Definition 3.1 (Traces [Aca05a, Definition 8]) *The **trace** is an ordered, rooted tree that describes the execution of a program P on an input. Every node corresponds to a function call, and is labeled with the name of the function; its arguments; the values it read from memory; and the return values of its children. A parent-child relationship represents a caller-callee relationship.*

Definition 3.2 (Cognates and Trace Distance [Aca05a, Definition 12]) *Given two traces T and T' of a program P, a node $u \in T$ is a **cognate** of a node $v \in T'$ if u and v have equal labels. The **trace distance** between T and T' is equal to the symmetric difference between the node-sets of T and T', i.e., distance is $|T| + |T'| - 2|C|$ where C is the set of cognates of T and T'.*

Definition 3.3 (Monotone Programs [Aca05a, Definition 15]) *Let T and T' be the trace of a program with inputs that differ by a single insertion or deletion. We say P is* **monotone** *if operations in T happen in the same order as their cognates in T' during a pre-order traversal of the traces.*

The change-propagation algorithm relies on a priority queue to propagate the change in the correct order. The main theorem of Acar [Aca05b] states that for monotone programs, the time for change-propagation is the same as the trace distance if the priority-queue overhead can be bounded by a constant. For the theorem, we say that a program is **$O(f(n))$-stable** for some input change, if the distance between the traces T, T' of the program with inputs I and I', where I' is obtained from I by applying the change, is bounded by $O(f(n))$. Note that stability is symmetric: insertions and deletions are indistinguishable.

Theorem 3.4 (Update time [Aca05a, Theorem 34]) *If a program P is monotone under a single insertion/deletion, and is $O(f(n))$-stable, and if the priority queue can be maintained in $O(1)$ time per operation, then change-propagation after an insertion/deletion takes $O(f(n))$ time.*

4 Dynamic Quad-Tree Analysis

The remainder of the analysis is devoted to showing that under single-point insertions and deletions, our Parallel Quad-Tree Refinement algorithm is monotone and $O(\lg L/s)$-stable, and that using a standard priority queue will take $O(1)$ time per PQ operation under these updates.

4.1 Monotonicity

Before proceeding to establish monotonicity, we must first detour to noticing that the same unbalanced cell can be added to the queue repeatedly, by several neighbours; across traces, it may be added by the same neighbour but in a different round. To sidestep these issues, we tag the the ADDWORK call with distinguishing information: the name of the cell that witnessed the imbalance, and the number of the round.

Throughout this section, T_0 and T_1 are two traces of QUADTREEREFINE; u and v are nodes of T_0, with round-pair r and r' respectively; and finally $\bar{u}$ and $\bar{v}$ are their cognates in T_1 (if any). We need to prove that if $u \prec v$ then $\bar{u} \prec \bar{v}$.

Lemma 4.1 *Trace nodes from different rounds are processed in monotone order across traces.*

Proof Given that u and v are cognates, they share round r; similarly $\bar{u}$ and $\bar{v}$ share round r'. Since u precedes v, $r < r'$. ■

The only question remaining is the order of items within a round r. We show by an inductive argument that it is also monotone:

Lemma 4.2 *Trace nodes from the same round occur in monotone order across traces.*

Proof The order of trace nodes within a round is defined by the order of cells on the W_l queue being processed. The order of cells in round 0 is clearly monotone: there is only one initial cell to split. Inductively, assume all cells in all prior rounds were processed monotonically between traces T_0 and T_1. Then their corresponding SPLITS were called in the same order in both traces. Therefore, the children generated by the splits were processed (in SPLITALL) in the same order in both traces. Finally, their corresponding ADDWORK calls occurred in the same order in both traces. Note that this last statement uses the fact that we only count as cognates ADDWORK calls with the same causer. ■

4.2 Trace Stability

Assume the inputs to traces T_0 and T_1 differ only in that trace T_1 has one additional point p. We want to show that only $O(\lg L/s)$ trace nodes differ between the two traces. There are two interesting kinds of nodes: ADDWORK and point relocation (from Line 10 of SPLIT). Any other kind of node is in one-to-one correspondence with an ADDWORK node, so counting those two types is sufficient.

Lemma 4.3 *The set of* ADDWORK *calls is $O(\lg L/s)$-stable.*

Proof If an ADDWORK call is being executed in T_1 but not T_0, then a cell c was split, which then caused the algorithm to find either (a) a child c_i of c is crowded that was not previously crowded, or (b) a neighbour c'_i of c is unbalanced that was not previously unbalanced. In case (a), c_i either contains p or is a neighbour of a cell that contains p. Thus c_i can be blamed on p. In case (b), either c was a newly crowded cell (in which case c is blamed on p as per case (a)), or c was unbalanced by another cell c'. By induction, c' must have been blamed on p, and so c is. Therefore, c'_i is blamed on p.

Any cell c is only named in $O(1)$ calls to ADDWORK: if c is crowded, there is exactly one call; if c is unbalanced, there may be up to one per neighbour, but there are $O(1)$ neighbours of c throughout the life of the algorithm.

Finally, Lemma 2.6 shows that p can only be blamed for $O(\lg L/s)$ splits; thus it can only be blamed for $O(\lg L/s)$ new calls to ADDWORK.

Conversly, an ADDWORK call executed in T_0 but not in T_1 can only be because the corresponding cell was split in T_1 earlier than in T_0. We know can blame the earlier split on p, and again this can only happen $O(1)$ times since c has $O(1)$ neighbours. ■

Lemma 4.4 *Point relocation work is $O(\lg L/s)$-stable.*

Proof Every point is reassigned at most $O(\lg L/s)$ times during the algorithm. Therefore, the computation to reassign the point p being added or removed is $O(\lg L/s)$-stable.

There are two reasons a point can be reassigned: either it is in a crowded cell being split, or it is in an uncrowded but unbalanced cell being split. A reassignment due to a crowded cell c can only be re-executed if the point p was either in the cell c or in a neighbour c' of c. Furthermore, we know that there was exactly one other point in c or c' — otherwise the algorithm would split regardless of the presence or absence of p, and the SPLIT call would be a cognate, not re-executed. Meanwhile, any unbalanced cell can only reassign at most one point.

In other words, if a split reassigns any points, it reassigns exactly one point. The set of splits is $O(\lg L/s)$-stable, and thus so is the set of point reassignments. ■

4.3 Priority Queue costs

Finally, we need to show that only $O(1)$ trace nodes are in the change propagation priority queue at any time. We know from prior proofs that during change propagation, only $O(1)$ trace nodes are processed in any size class. Furthermore, at most 3 size classes are in the queue at any one time: the current size class; unbalanced cells in one size class larger, if any; and crowded cells which may be in a smaller size class.

4.4 Main Result

Theorem 4.5 *The* QUADTREEREFINE *algorithm, sequentialized and dynamized as described, can maintain a balanced quad-tree over a point set in any fixed dimension under any sequence of single-point additions* *and removals. Let $\mathcal{P}_0$ and $\mathcal{P}_1$ be the point sets before and after an update; let $s = \min(s_0, s_1)$ and $n = \max(|\mathcal{P}_0|, |\mathcal{P}_1|)$. Then our dynamic algorithm runs in time $O(\lg L/s)$ and uses a history-independent data structure of size $O(n \lg L/s)$.*

Proof The Lemmata of the present Section show that QuadTreeRefine is $O(\lg L/s)$-stable, monotone, and can be dynamized using a constant-time priority queue for the change propagation algorithm. Therefore, Theorem 3.4 applies and yields the time bound and history independence.

History-independence further implies that the data structure is topologically identical to one that results from inserting the n points of $\mathcal{P}_1$ one by one. Give our time bound, we know that we can do n insertions in $O(n \lg L/s)$ time. Clearly, we cannot use more space than time, which yields our space bound. ■

5 Generating small meshes in 2d

The meshes output by the postprocess described in Section 2.2 are within a constant factor of optimal size and of the best possible quality. In practice however, they are substantially larger than than those output by Ruppert refinement [Rup95], and unlike in Ruppert refinement, they do not offer the user of the mesh any control of the desired quality bound. Üngör [Üng04] described a way of choosing what he called an *off-center*: given a bad-quality triangle (one with a small angle), we can insert a Steiner point so that the shortest edge of the triangle forms a triangle with the off-center that exactly achieves the quality threshhold. In theory, off-centers yield optimal-size meshes. In practice, off-center meshes are the smallest known. Har-Peled and Üngör [HPÜ05] then showed how to use off-centers to post-process a balanced quad-tree in order to simultaneously achieve the time bounds from quadtree meshing and the small output size from off-center meshing. We show here how to dynamize the Har-Peled and Üngör postprocess. Due to space constraints, we leave the full details to the Appendix A.

The algorithm proceeds as follows: iteratively, in order from smallest to largest quadtree cell, the algorithm considers every input point p in a given cell, then searches neighbouring cells for an input point q. Having found such a pair of points, it checks whether there is a third point r such that pqr is a Delaunay triangle, and pqr has good quality. If there is no such r, then pq is a termed *loose* pair. The algorithm constructs an appropriate r using the off-center, and inserts this r, which is now treated as an input point. During this routine, the quadtree serves the purpose of performing the point location (for p and q) and range queries (for r, if it exists).

As a final post-process, we can again use the technique of starting from the smallest cell to the largest and using the quadtree for point location to compute the Delaunay triangulation in linear time.

We deviate in only one respect from the original algorithm of Har-Peled and Üngör: they left undefined the order of operations pairs within a size class. To establish our stability bounds, we require that they be done in FIFO order. This should be reminiscent of our modification of the Bern *et al.* algorithm.

Given that our algorithm performs the same steps as the original algorithm, the correctness, size optimality (and in-practice performance), and static runtime of our modified HPU algorithm immediately follow. Dynamic stability is all that is left to establish. The argument (detailed in the Appendix) is reminiscent of the dynamic stability argument for the quadtree itself: we define a notion of *blame* for off-centers upon input points, and prove a packing lemma:

Lemma 5.1 (Off-centers pack) *Let r be an off-center that blames an input point p. Then $|rp| \in \Theta(NN(r))$ where $NN(r)$ is the nearest neighbour of r when r is inserted.*

Theorem 5.2 *Given a dynamic point set $\mathcal{P} \in [1/3, 2/3]^2$ and a radius/edge ratio $\rho > 1$, we can dynamically maintain a mesh of the desired quality using within a (in practice small) constant factor of the optimal number of Steiner vertices. Each addition to or deletion from the input point set can be performed in $O(\lg L/s)$ time.*

Proof Using self-adjusting computation, run the dynamically-stable quadtree algorithm described earlier, and use that as input to the dynamically-stable HPU postprocess described in this section. Upon a point addition or deletion, we know from Theorem 4.5 that the quadtree updates in $O(\lg L/s)$ time. Each cell is only read $O(1)$ times by the postprocess, so propagating the quadtree changes through the postprocess is fast. Finally, HPU is itself $O(\lg L/s)$-stable, by the previous packing lemma. We omit the monotonicity and priority queue arguments for brevity. ■

6 Conclusions

In this paper, we showed a dynamic algorithm, for maintaining a balanced quadtree in arbitrary dimension. The algorithm is optimal for a large class of inputs (inputs with polynmomial spread), yields size-optimal meshes, and is history-independent. The algorithm is obtained by applying self-adjusting-computation techniques to dynamize an algorithm for generating quadtrees. We believe this is the first dynamic algorithm for computing meshes in optimal time. As a second result, we gave a dynamization of the Har-Peled and Üngör technique for postprocessing quadtrees [HPÜ05] to obtain size-optimal meshes that are the smallest known in practice. Based on the history independence, and composibility properties of our dynamic quadtree algorithm, composing the two results yield a technique for dynamically generating and maintaining practically small 2-d meshes in optimal time for inputs with polynomial spread. Since meshes are required in many application domains, we expect that our results will find applications in a number of aread (e.g., scientific computing, CAD design, and streaming/out-of-core meshing).

The algorithm that we give here only handles inputs points but not input-features such as segments or polygons. Even in the static case, handling input features is difficult: the first time-optimal algorithm that can handle features was discovered very recently [HMP06, HMP07]. As with the quadtree algorithm, this algorithm has a data dependency depth of $O(\lg L/s)$. We therefore hope to be able to use the techniques in this paper to dynamize that algorithm and thus handle more complicated geometries.

References

[ABBT06] 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 Programming Language Design and Implementation*, 2006.

[ABT07] Umut A. Acar, Guy E. Blelloch, and Kanat Tangwongsan. Kinetic 3d convex hulls via self-adjusting computation (an illustration). In *ACM Symposium on Computational Geometry (SCG)*, 2007.

[ABTV06] Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Jorge L. Vittes. Kinetic algorithms via self-adjusting computation. Technical Report CMU-CS-06-115, Department of Computer Science, Carnegie Mellon University, March 2006.

[Aca05a] Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[Aca05b] Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[BA76] Ivo Babuška and A. K. Aziz. On the Angle Condition in the Finite Element Method. *SIAM Journal on Numerical Analysis*, 13(2):214–226, April 1976.

[Bak01] Timothy J. Baker. Mesh movement and metamorphosis. In *10th International Meshing Roundtable*, pages 387–396, 2001.

[BEG90] Marshall Bern, David Eppstein, and John R. Gilbert. Provably Good Mesh Generation. In *31st Annual Symposium on Foundations of Computer Science*, pages 231–241. IEEE Computer Society Press, 1990.

[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 89–983, Department of Computer Science, Cornell University, 1989.

[EGS05] David Eppstein, Michael T. Goodrich, and Jonathan Zheng Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In *21st Symposium on Computational Geometry*, pages 296–305, 2005.

[HMP06] Benoît Hudson, Gary Miller, and Todd Phillips. Sparse Voronoi Refinement. In *Proceedings of the 15th International Meshing Roundtable*, pages 339–356, Birmingham, Alabama, 2006.

[HMP07] Benoît Hudson, Gary L. Miller, and Todd Phillips. Sparse Parallel Delaunay Refinement. In *19th ACM Symposium on Parallelism in Algorithms and Architectures*, 2007.

[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.

[Joh87] Claes Johnson. *Numerical solutions of partial differential equations by the finite element method*. Cambridge University Press, 1987.

[LTÜ98] X.-Y. Li, S.-H. Teng, and A. Üngör. Simultaneous refinement and coarsening: adaptive meshing with moving boundaries. In *7th International Meshing Roundtable*, pages 201–210, Dearborn, Mich., 1998.

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

[MV92] Scott A. Mitchell and Stephen A. Vavasis. Quality Mesh Generation in Three Dimensions. In *Proceedings of the Eighth Annual Symposium on Computational Geometry*, pages 212–221, 1992.

[MV00] Scott A. Mitchell and Stephen A. Vavasis. Quality mesh generation in higher dimensions. *SIAM Journal on Computing*, 29(4):1334–1370, 2000.

[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.

[Ove81] Mark H. Overmars. Dynamization of order decomposable set problems. *J. Algorithms*, 2(3):245–260, 1981.

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

[She98] Jonathan Richard Shewchuk. Tetrahedral Mesh Generation by Delaunay Refinement. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, pages 86–95, Minneapolis, Minnesota, June 1998. Association for Computing Machinery.

[Üng04] Alper Üngör. Off-centers: A new type of Steiner point for computing size-optimal quality-guaranteed Delaunay triangulations. In *LATIN*, pages 152–161, 2004.

```
DynHPU($\mathcal{P} \in [1/3, 2/3]^2$, $\rho$)
Construct a balanced quadtree $\mathcal{QT}$
Rescale so that the size of the smallest cell is 1; let $L$ be the largest cell.
for ($i = 0$ to $\lg L$) do
    enqueue all cells of size $2^i$ into $Q_i$
for ($i = 0$ to $\lg L$) do
    while ($Q_i$ is non-empty)
        collect all loose pairs $pq$ where $p$ is an active vertex in a cell on $Q_i$
        empty $Q_i$
        for each collected $pq$
            if $pq$ is no longer loose then skip $pq$
            compute the off-center $r$ of $pq$
            add $r$ to the smallest cell $c$ such that (a) $c$ contains $r$, (b) $|c| \geq 2^i$, (c) $c_{\mathrm{low}}|c| \leq \|pr\| \leq c_{\mathrm{up}}|c|$
            append $c$ to $Q_{\lg|c|}$
            if $pq$ is still loose, repeat
```

Figure 3: A dynamically-stable version of the Har-Peled and Üngör [HPÜ05] algorithm. The key difference is that we define more carefully the ordering of items on the work queue. We also require the use of a dynamically-stable balanced quadtree algorithm such as DynQT. Note that Line 14 is triggered only if pq is loose from both left and right.

A Generating small meshes in 2d

We use the following terms from Har-Peled and Üngör. Most of the following definitions define an orientation; we write the definitions for the counterclockwise (ccw) orientation and leave the reader to perform appropriate substitutions to define the clockwise (cw) equivalent.

Definition A.1 (Leaf) *Given a pair of points p and q, take a point c such that $|cp| = |cq| = \rho|pq|$, and $|pqc|$ forms a counterclockwise cycle. The **ccw-leaf** of pq is the disc $D(c, \rho|pq|)$.*

Definition A.2 (Loose pair) *A pair pq is **ccw-loose** if the **ccw-leaf** is empty of any points. A pair pq is **loose** if it is either **ccw-loose** or **cw-loose**.*

Definition A.3 (Crescent) *Given a pair pq, let c be the point on the ccw-leaf of pq that is farthest from p and q. The **ccw-crescent** of pq is the portion of the disc $D(c, |pc|)$ with the ccw-leaf removed.*

Definition A.4 (Off-center) *Let pq be a ccw-loose pair pq. If the ccw-crescent of pq is empty, then the **ccw-offcenter** of pq is the point c from the definition of the crescent. If the ccw-crescent is non-empty, take the point p' such that disc that circumscribes p, p', and q is empty. The **ccw-offcenter** is the center of that disc.*

Definition A.5 (Active point) *A point p is **active** if it may form a loose pair with another active point. See [HPÜ05, Lemmata 4.8–4.11] for proofs and technical definitions. Only $O(1)$ points are active in any cell of a balanced quadtree.*

We present our modification of the Har-Peled and Üngör algorithm in Figure 3. DynHPU takes as input the point set, a radius/edge quality bound $\rho > \sqrt{2}$, and a dynamic quadtree. It produces as output a list of points. We can use a modification of DynHPU to produce the Delaunay triangulation in time linear in the output size: to decide that a pair pq is not loose requires finding a point t in the leaf of pq such that pqt is Delaunay.

The algorithm proceeds as follows: iteratively, roughly in order from smallest to largest loose pair, the algorithm identifies a loose pair and inserts its off-center (or both off-centers, if it is loose from both sides). It uses the quadtree for two purposes: to order the loose pairs (to within a constant factor), and to test for looseness. We deviate in one respect from the original algorithm of Har-Peled and Üngör: they left undefined the order of loose pairs within a size class i (Lines 7–15), whereas to establish Lemma A.8 we require that they be done in FIFO order. In essence, we simulate processing Q_i in parallel.

Given that our algorithm performs the same steps as the original algorithm, the correctness, size optimality (and in-practice performance), and static runtime of our DynHPU algorithm immediately follow. Dynamic stability is all that is left to establish. The argument will be reminiscent of the dynamic stability argument for DynQT: we show that any input point p can only be blamed on $O(1)$ off-center insertions for any value of i.

Definition A.6 (Insertion radius) *The **insertion radius** of an off-center r, denoted $\mathrm{IR}(r)$, is the distance from r to its nearest neighbour at the time r was inserted.*

Lemma A.7 (The insertion radius is large) *Consider a loose pair pq and their off-center r. Then the insertion radius of r follows $2\rho|pq| > \mathrm{IR}(r) \geq \rho|pq|$.*

Proof There are two cases: (1) if there is a vertex t in the crescent, then r is the circumcenter of pqt. By definition, pqt is Delaunay: its circumdisc is empty of any other points. Therefore, $\mathrm{IR}(r) = R(pqt)$. Also, because pq is loose, pqt must have bad radius/edge ratio: $R(pqt)/|pq| > \rho$, or equivalently $\mathrm{IR}(r) > \rho|pq|$.

If instead the crescent is empty, then r is the farthest point on the flower of pq, and we know that the crescent of pq is empty of points. The crescent of pq has radius $|pr|$, which shows that $\mathrm{IR}(r) = |pr|$. From the Pythagorean theorem, we can compute $\mathrm{IR}(r) = |pr| > \rho|pq|$.

In either case, r, p, and q all lie on a circle of radius at most $\rho|pq|$, and thus can be separated by no more than twice that distance. ■

Lemma A.8 (Loose pairs grow geometrically) *After every iteration of the* DynHPU *while loop, the size of the smallest remaining loose pair in iteration i of the for loop grows by a factor at least ρ.*

Proof Let s_{ij} be the length of the shortest loose pair at the beginning of the jth iteration of the while loop in iteration i of the for loop. Consider a loose pair seen at the end of iteration ij, but not seen at the beginning of the iteration. Such a loose pair must include at least one new off-center r; if it is a pair made of two new off-centers, let r be the newer one. That off-center issued from a loose pair of length at least s_{ij}. By Lemma A.7, the nearest neighbour of r is at distance at least ρs_{ij}; in particular, its partner in the loose pair must be at least that far. ■

Lemma A.9 (Loose pairs don't grow too fast) *All loose pairs processed in iteration i of the for loop have length in $\Theta(2^i)$.*

Proof The upper and lower bounds were proven before [HPÜ05, Lemmata 4.3, 4.7]. ■

Definition A.10 (Blame for off-centers) *An off-center r **directly blames** a point p if r issues from a loose pair around p. Transitively, r **indirectly blames** those that p blames.*

Lemma A.11 (Off-centers pack) *Let r be an off-center that blames a point p. Then $|rp| \in \Theta(\mathrm{IR}(r))$.*

Proof That $\mathrm{IR}(r) \leq |rp|$ is trivial: the insertion radius of r is empty of points.

If r directly blames p, then this is restating Lemma A.7.

If r directly blames a point q that transitively blames p, then by the triangle inequality, we have $|rp| \leq |rq| + |qp|$. We know that $|rq| = \mathrm{IR}(r)$ by definition. We can inductively assume that there is a constant k such that $|pq| \leq k\,\mathrm{IR}(q)$. Thus, $|rp| \leq \mathrm{IR}(r) + k\,\mathrm{IR}(q)$. It remains to bound $\mathrm{IR}(q)$ in terms of $\mathrm{IR}(r)$; this follows from Lemma A.7. Thus, $|rp| \leq (1 + k/\rho)\,\mathrm{IR}(r)$. For any $\rho \geq 1$, k is a constant with $k = \rho/(\rho - 1)$. ■

Finally, we can state the overall result:

Theorem A.12 *Under self-adjusting computation,* DynHPU *runs in $O(\lg L/s)$ time per addition to or removal from the input point set.*

Proof By Theorem 4.5, maintaining the dynamic quad tree takes $O(\lg L/s)$ time per update.

Using Lemma A.11 in an area packing argument, at most $O(1)$ off-centers in iteration i blame any input point p. Therefore, at most $O(\lg L/s)$ off-centers of any iteration blame p. Every off-center insertion reads at most $O(1)$ input or Steiner points, and $O(1)$ cells of the quadtree.

For brevity, we elide the monotonicity argument, which is essentially identical to that in Section 4.1.

Again using the fact that every while loop iteration is $O(1)$-stable, and using the fact (derived from Lemma A.9) that we only affect $O(1)$ iterations of DynHPU at a time, the priority queue costs of DynHPU are $O(1)$ per operation. ■