Using Bistellar Flips for Rotations in Point Location Structures

Benoît Hudson Gary Miller*
Carnegie Mellon University
Pittsburgh, PA 15213
{bhudson,glmiller}@cs.cmu.edu

Abstract

Point location in dynamic Delaunay triangulations is a problem that as yet has no elegant solution. Current approaches either only give guarantees against a weakened adversary, or require superlinear space. In this paper we propose that we should seek intuition from balanced binary search trees, where rotations are used to maintain a shallow worst-case depth. We describe a (well-known) data structure and a novel and simple algorithm based on bistellar flips to implement rotation in the structure. The rotation takes time linear in the change in the data structure. The hope is to provide a tool that would lead to the design of an efficient dynamic Delaunay point location data structure.

1 Introduction

A major cost in Delaunay mesh refinement algorithms, where a Delaunay triangulation is refined by adding new points to achieve some quality objective (large minimum angle, for instance), is the subroutine to discover which triangle contains a given candidate point. We are therefore interested in finding fast, simple algorithms to perform this point location operation. In addition, we need to be able to maintain the point locator through insertions (as we refine the mesh) and deletions (as we coarsen it). While the problem generalizes trivially to any dimension, we are for the time being mainly interested in handling point sets in the plane.

In the static case, Kirkpatrick [6] has a deterministic data structure with $O(n)$ size that answers queries in $O(\log n)$ time. Construction takes $O(n \log n)$ time.

However, in the dynamic case, no such bound is known. Under the so-called *communist model* [8], Mulmuley has data structures that offer expected $O(n)$ size data structures and expected $O(\log n)$ time updates and $O(\log^2 n)$ time queries (the update time bound assumes we have a pointer into the structure, perhaps by having done a query beforehand). Also, Devillers *et al.* [2] have a data structure that offers expected $O(n)$ size expected $O(\log n)$ time updates and queries; Clarkson *et al.* [1] have equivalent bounds. However, the adversary is greatly weakened in this model: when it wants to insert or remove a vertex, it must choose a random vertex! Against the usual adversary, all these data structure degenerate to worst case $\Omega(n)$ depth and $\Omega(n^2)$ size.

One difficulty in designing dynamic data structures for Delaunay point location is that any data structure that explicitly represents the triangulation has a lower bound of $\Omega(n)$ per update, even in an amortized setting: the adversary can change all the triangles with every insertion if it inserts points on a parabola from right to left. Assuming we limit ourselves to data structures that represent the triangulation explicitly, this indicates that we want a data structure with output-sensitive update times.

In that vein, we can use the deterministic data structure of Goodrich and Tamassia [4] to obtain $O(k \log n)$ update time, with $O(n \log n)$ space and $O(\log^2 n)$ query time, where k triangles have been modified.

The open problem this work attempts to attack is: can we design a data structure that achieves the optimal $O(n)$ space and $O(\log n)$ query time, and fast output-sensitive update time – ideally $O(k + \log n)$? We do not solve the problem, but we present tools that may prove to be useful to analyze it. Our tools are only proven to work for Delaunay triangulations in the plane, but most of the techniques used easily extend to higher dimensions, so we expect the entire approach can be generalized.

2 Approach

The main intuition in this paper is that to solve the problem in two dimensions, it may be useful to draw on our experience with the analogous problem in one dimension. In one dimension, the Delaunay triangulation is a set of 1-simplices (segments) defined by two points at its extremities, and containing no point within its circumcircle (the same segment). In other words, a query is: given a point on the line, tell me what interval it lies in. The usual approach for this is to use a balanced binary search tree: AVL, red-black, splay, treap, and likely others. All these approaches share two essential features: a binary search tree as the underlying data structure, and a method of rebalancing that transforms the tree using only the rotation operation. A tree induces a partial order on the insertion times of the point it contains. Rotation corresponds to inverting the insertion order of two points a and b: before rotation, a is inserted before b; after rotation, b precedes a in the new induced partial order. The two points

*This work was supported in part by the National Science Foundation as part of the Aladdin Center (www.aladdin.cmu.edu) and the Sangria Project (www.cs.cmu.edu/ sangria) under grants ACI-0086093 and CCR-0122581.

must be adjacent: there cannot be another point c with insertion time between a and b.

Sleator *et al* [9] prove that doing rotations on trees of n nodes is isomorphic to doing bistellar flips on triangulations of convex $(n+2)$-gons. Our conjecture is that this generalizes: doing rotations in a search data structure for Delaunay triangulations is isomorphic to doing series of bistellar flips on a class of triangulations in one higher dimension. This paper demonstrates the isomorphism for planar Delaunay triangulations.

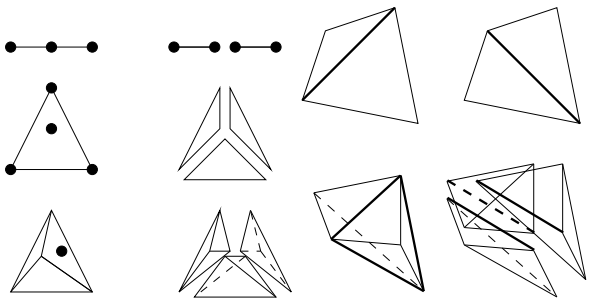
Our flip-based rotation operation runs in time linear in the size of the change of the data structure (see Corollary 3), which in general is faster than simply tearing out the parts of the data structure related to a and b and retriangulating. This is critical in Delaunay triangulation in the plane and in higher dimensions, because the size of the part of the data structure related to any one vertex can be linear in the number of vertices. Using bistellar flips makes the code simple, because the usual approach to building a Delaunay triangulation already requires implementing them; the rotation algorithm itself is also simple. Finally, by basing the rotation operation on small atomic flips, the data structure is always valid, which allows for parallel access to it.

3 Tools

We use three main tools in our algorithms and analysis. On a first reading of the paper, it may be worthwhile to skim through the following sections and return to them when their usefulness has been fully motivated.

3.1 Bistellar Flips

Bistellar flips are a topological transformation defined on triangulations in any dimension d. There are essentially two types: ones that introduce or remove a vertex from the triangulation, and ones that flip faces of the triangulation.



A Order 0 flips **B** Order 1 flips

Figure 1: The bistellar flips in dimensions 1, 2, and 3.

A non-degenerate geometric bistellar flip – henceforth, "bistellar flip" – of order i takes as input a $(d-i)$-simplex e (an edge, for instance). Let C be the set of d-simplices that include e. If C forms a convex space and has $d+2$ vertices, then a theorem of Lawson [7] implies that there are exactly two triangulations of the vertices: one that includes e and one that instead includes the i-simplex made up from the other vertices. A bistellar flip moves between the two triangulations. A flip of order i reverses a flip of order $(d-i)$, so we only discuss the flips of order less than $d/2$.

Not every simplex can be eliminated by a single bistellar flip: if C is not convex, Lawson's theorem does not apply; if C is too small (for instance, in two dimensions, one triangle), there is nothing to flip; if C is too large (only possible in $d \geq 3$) then the bistellar flip is not defined, although there may exist a series of flips that will eventually eliminate e.

Figure 1 shows the bistellar flips in 1, 2, and 3 dimensions. In $d=1$, a d-simplex is a segment. The order-0 flip breaks a segment in two by introducing a new vertex.

In $d=2$, a d-simplex is a triangle. The order-0 flip again introduces a vertex, splitting a triangle into three. There is also a order-1 flip, which flips an edge shared by two triangles, creating two new triangles.

In $d=3$, a d-simplex is a tetrahedron. The order-0 flip now splits a tetrahedron into four. The order-1 flip takes two tetrahedra that share a triangular face t, and pierces t with the segment between the apexes of each tetrahedron, producing three tetrahedra along that edge (the face and edge are shown in bold in the figure).

Because the order-1 flip in two dimensions takes two triangles to two triangles, it is often called a 2–2 flip. Similarly, the order-1 flip in three dimensions is a 2–3 flip, while the order-2 flip is a 3–2 flip.

3.2 Parabolic lifting map

The parabolic lifting map takes points in $\mathcal{R}^d$ to the paraboloid in $\mathcal{R}^{d+1}$. That is, a point $p = \langle p_1, \ldots, p_d \rangle$ is mapped to $p^+ = \langle p_1, \ldots, p_d, z = \sum_{i=1}^{d} p_i^2 \rangle$. The z axis gives us a well-defined notion of up and down in arbitrary dimensions.

We study the parabolic lifting map because it exposes a connection between the Delaunay triangulation and the lower convex hull on the paraboloid. Given a set $S \subset \mathcal{R}^d$, the $(d+1)$-dimensional lower convex hull of S^+ projected back down to $\mathcal{R}^d$ is the Delaunay triangulation of S. Figure 2 shows this process for $d=1$.

3.3 The history DAG

Insertion of a new vertex v into an existing Delaunay triangulation in d dimensions can be done by the following algorithm, paraphrased from Edelsbrunner and Shah [3]. Consult their paper for a proof of its correctness (and of its running time).

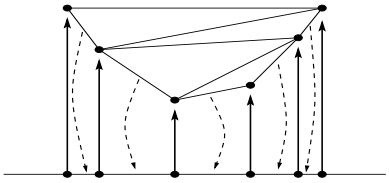


Figure 2: The parabolic lifting map in one dimension. First, we lift the points on the line to the parabola. Then we take the convex hull, and project the facets (segments) of the convex hull back down to the line. The history DAG is also displayed for one insertion order.

INSERT-VERTEX(v)
1. find the d-simplex s that contains v
2. split s into $d+1$ simplices using a 0-order bistellar flip
3. while a neighbouring d-simplex s' contains v in its circumcircle,
3.1 find a convex set of d-simplicies including s'
3.2 destroy s' by performing the flip

Seed the algorithm by creating a large d-simplex in which all the points will fit. This is for simplicity of exposition, and is not fundamental.

We can maintain a data structure that describes the history of the run of the algorithm: upon splitting a simplex s, set a pointer from t to the $d+1$ new simplices that replace it. Similarly, upon flipping a simplex s' in step 3.2, set a pointer from the simplicies that were destroyed to the simplicies that were created by the flip.

Since this data structure is acyclic and represents the history of the algorithm, it is termed the *history DAG* [5]. We say a simplex is *buried* by the simplex that replaced it. We say a simplex is a *leaf* if no simplex buries it – notice that a leaf is a Delaunay facet. Finally, the bounding simplex is the *root*.

The history DAG can be used to implement the search for the simplex that contains v. Starting with s being the root, we look at the simplices that buried s and recur into the single simplex that contains v, returning the leaf we come to.

3.3.1 In $d = 1$ and $d = 2$

In dimension $d = 1$, a d-simplex is a segment. A segment on the line can only have a point inside its circumcircle if the point is inside the segment, therefore when inserting points into the 1-d Delaunay triangulation, the algorithm will only perform an order-0 flip, and line 3.2 will never be invoked. The history DAG will thus only have pointers from buried segment to their two subsegments. In other words, the history DAG in this case is a normal binary search tree.

In dimension $d = 2$, a d-simplex is a triangle. Splitting a triangle creates three new triangles that partition the space of s. Flips are important now; they take 2 triangles and flip the edge between them to create two new triangles. This means that the structure is no longer a tree, although it remains acyclic.

3.3.2 Lifting the history DAG

The history DAG has an elegant interpretation in the lifting map. Splitting a face (a d-simplex) corresponds to creating a $(d+1)$-simplex with one face (the face that was split) on the top, and the other faces (the $d+1$ new faces) on the bottom. Similarly, flipping a non-Delaunay d-simplex corresponds to taking a valley whose walls are formed by the old simplicies and covering it with the new simplicies, forming a $(d+1)$-simplex from the filled-in valley. Thus the nodes of the history DAG correspond to simplices in a simplicial complex that fills the convex closure of the set of points lifted to the paraboloid.

In $d = 2$, the set of tetrahedra created by inserting a vertex v defines a shape not unlike a circus pavillion: some number of poles holding up canvas, meeting in a point. Because of this, we call this set the $\mathrm{tent}(v)$. Using the same analogy, the triangles that were on the surface before v but are buried by the $\mathrm{tent}(v)$ are called the $\mathrm{base}(v)$. These definitions trivially generalize to arbitrary dimension.

4 Rotations in 2-d

It's already been known for almost 20 years that rotation in a tree (the 1-d Delaunay point location structure) is a bistellar flip in one higher dimension. Here we show that rotation in the 2-d Delaunay point location structure – the history DAG – is a series of bistellar flips in three dimensions. There are two main differences between the 1-d and the 2-d cases: one point now corresponds to a set of simplices (a *tent*) in our DAG; and our DAG is not a tree. In one dimension, vertices a and b can be rotated if they are parent and child. In two dimensions, we can rotate $\mathrm{tent}(a)$ and $\mathrm{tent}(b)$ if some tetrahedra of $\mathrm{tent}(b)$ lie over faces of $\mathrm{tent}(a)$, but only if no other $\mathrm{tent}(c)$ is overlain by $\mathrm{tent}(b)$ while overlying $\mathrm{tent}(a)$.

A key observation is that during the rotation operation, nothing happens to simplices outside of $\mathrm{tent}(a) \cup \mathrm{tent}(b)$. Therefore, the *surface* of the rotation region (the exposed lower faces – initially, the lower faces of $\mathrm{tent}(b)$, plus the lower faces of $\mathrm{tent}(a)$ that $\mathrm{tent}(b)$ does not bury) does not change; and similarly with the *base* of the rotation region.

In the algorithm, Q is a structure that allows the constant-time operations `push` to add an element, `pop` to remove an arbitrary element, and `remove` to remove a given element. Despite the naming, Q need not be a queue or a stack: ordering is not important.

ROTATE(a, b)
1. $Q \leftarrow$ all tetrahedra in $\mathrm{tent}(b)$ that have both a and b as vertices
2. while Q not empty
2.1 pop T from Q
2.2 if T has a legal bistellar flip (see below)
2.2.1 let C be the tetrahedra in the flip
2.2.2 remove all tetrahedra C from Q if present
2.2.3 perform the bistellar flip $C \to C'$
2.2.4 push onto Q the tetrahedra of C' that have both a and b

In step 2.2, to find a legal flip for a tetrahedron T, we form all combinations of T and its neighbours. In a legal flip C, all tetrahedra have a, at least one (namely, T) has b, the tetrahedra fill their convex hull, and all tetrahedra are within the rotation region. We can check the last restriction by numbering vertices by an insertion order that yields the DAG. A tetrahedron that has a vertex numbered higher than a or b is below the rotation region and thus cannot be involved in a valid flip. A tetrahedron above the rotation region cannot have a.

For lack of space, we present here only a sketch of the correctness of the approach; the detailed proofs are in the full paper at `www.cccg.ca`.

- **Proposition 1** *Progress: Every flip we perform produces at least one tetrahedron of the final result.*
- **Proposition 2** *Non-stick: Unless we have reached the end of the algorithm, every flip brings us to a state in which another flip can be done.*

Together, these two propositions mean that the algorithm terminates (since there is a finite amount of work to be done) with the correct answer (since it does not stop unless all of the work has been done). As a corollary to Proposition 1, we can compute the running time:

Corollary 3 *The algorithm terminates after doing a number of flips equal to the number of triangles in* $\mathrm{base}(a)$ *on which* b *encroaches.*

The only step that takes more than constant work per flip is the first, intializing Q: it may take time $O(|\mathrm{tent}(b)|)$ even if there are few flips to be done. However, if we give the algorithm one tetrahedron that has both a and b on it, the algorithm can search the DAG from there in time linear in the number of tetrahedra that have a and b. All of these tetrahedra will have to be flipped, except perhaps for one, so the total time is linear in the number of flips, which is linear in the change in the data structure.

5 Designing rotation logic

With the machinery we have brought to bear on this problem, we are now in a good position to begin considering not only *how*, but *when* and *where* to perform rotations. Unfortunately, a completely straightforward application of binary tree techniques may not directly work. In particular, we have investigated implementing a treap analogue.

A treap is a binary search tree with the added property that every point inserted into the tree gets a random priority. The priorities are kept in heap order via rotations. Equivalently, the priority is a virtual insertion time, and the tree is made to be indistinguishable from having inserted the points in the random order. Because a search tree built in random order has $O(\lg n)$ depth, so does the treap.

We can easily generalize this to the Delaunay history DAG and our rotation operation – indeed, we have implemented this. However, it is not true that a Delaunay DAG built in random order is shallow: in fact, there are point sets on which the expected depth is linear.

Despite this negative result, we know that there is an insertion order that induces a shallow DAG: Kirkpatrick's algorithm produces one. Therefore, we expect that there is a rotation-based algorithm for maintaining a shallow Delaunay point location data structure.

References

[1] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. In *Symposium on Theoretical Aspects of Computer Science*, pages 463–474, 1992.

[2] O. Devillers, S. Meiser, and M. Teillaud. Fully dynamic delaunay triangulation in logarithmic expected time per operation. *Computational Geometry Theory and Applications*, 2(2):55–80, 1992.

[3] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. *Algorithmica*, 15:223–241, 1996.

[4] M. T. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. In *23rd Annual ACM Symposium on the Theory Computing*, pages 523–533, 1991.

[5] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized Incremental Construction of Delaunay and Voronoi Diagrams. *Algorithmica*, 7:381–413, 1992.

[6] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal of Computing*, 12(1):28–35, 1983.

[7] C. L. Lawson. Properties of n-dimensional triangulations. *Computer Aided Geometric Design*, 3(4):231–246, 1987.

[8] K. Mulmuley. *Computational Geometry*. 1994.

[9] D. Sleator, R. Tarjan, and W. Thurston. Rotation distance, triangulations and hyperbolic geometry. *Journal of the American Mathematical Society*, 1(3):647–681, 1988.