Chapter 5

Algorithms for constructing Delaunay triangulations in $\mathbb{R}^3$

The most popular algorithms for constructing Delaunay triangulations in $\mathbb{R}^3$ are incremental insertion and gift-wrapping algorithms, both of which generalize to three or more dimensions with little difficulty. This chapter reprises those algorithms, with attention to the aspects that are different in three dimensions. In particular, the analysis of the running time of point location with a conflict graph is more complicated in three dimensions than in the plane. We use this gap as an opportunity to introduce a more sophisticated vertex ordering and its analysis. Instead of fully randomizing the order in which vertices are inserted, we recommend using a *biased randomized insertion order* that employs just enough randomness to ensure that the expected running time is the worst-case optimal $O(n^2)$—or better yet, $O(n \log n)$ time for the classes of point sets most commonly triangulated in practice—while maintaining enough spatial locality that implementations of the algorithm use the memory hierarchy more efficiently. This vertex ordering, combined with a simpler point location method, yields the fastest three-dimensional Delaunay triangulators in practice.

CDTs have received much less study in three dimensions than in two. There are two classes of algorithm available: gift-wrapping and incremental polygon insertion. Gift-wrapping is easier to implement; it is not much different in three dimensions than in two. It runs in $O(nh)$ time for Delaunay triangulations and $O(nmh)$ time for CDTs, where n is the number of vertices, m is the total complexity of the PLC's polygons, and h is the number of tetrahedra produced.

Perhaps the fastest three-dimensional CDT construction algorithm in practice is similar to the one we advocate in two dimensions. First, construct a Delaunay triangulation of the PLC's vertices, then insert its polygons one by one with a flip algorithm described in Section 5.8. This algorithm constructs a CDT in $O(n^2 \log n)$ time, though there are reasons to believe it will run in $O(n \log n)$ time on most PLCs in practice. Be forewarned, however, that this algorithm only works on edge-protected PLCs. This is rarely a fatal restriction, because a mesh generation algorithm that uses CDTs should probably insert vertices on the PLC's edges to make it edge-protected and ensure that it has a CDT.

Procedure	Purpose
A T (u, v, w, x)	Add a positively oriented tetrahedron *uvwx*
D T (u, v, w, x)	Delete a positively oriented tetrahedron *uvwx*
A (u, v, w)	Return a vertex *x* such that *uvwx* is a positively oriented tetrahedron
A 2V (u)	Return vertices *v*, *w*, *x* such that *uvwx* is a positively oriented tetrahedron

Figure 5.1: An interface for a three-dimensional triangulation data structure.

5.1 A dictionary data structure for tetrahedralizations

Figure 5.1 summarizes an interface for storing a tetrahedral complex, analogous to the interface for planar triangulations in Section 3.2. Two procedures, A T and D T -, specify a tetrahedron to be added or deleted by listing its vertices with a positive orientation, as described in Section 3.1. The procedure A recovers the tetrahedron adjoining a specified oriented triangular face, or returns $\emptyset$ if there is no such tetrahedron. The vertices of a tetrahedron may include the ghost vertex. The data structure enforces the invariant that only two tetrahedra may adjoin a triangular face, and only one on each side of the face.

The simplest fast implementation echoes the implementation described in Section 3.2. Store each tetrahedron *uvwx* four times in a hash table, keyed on the oriented faces *uvw*, *uxv*, *uwx*, and *vxw*. Then the first three procedures run in expected $O(1)$ time. To support A 2V queries, an array stores, for each vertex *u*, a triangle *uvw* such that the most recently added tetrahedron adjoining *u* has *uvw* for a face. As Section 3.2 discusses, these queries take expected $O(1)$ time in most circumstances, but not when the most recently added tetrahedron adjoining *u* has subsequently been deleted.

The interface and data structure extend easily to permit the storage of triangles or edges that are not part of any tetrahedron, but it does not support fast adjacency queries on edges.

5.2 Delaunay vertex insertion in $\mathbb{R}^3$

The Bowyer–Watson algorithm extends in a straightforward way to three (or more) dimensions. Recall that the algorithm inserts a vertex *u* into a Delaunay triangulation in four steps. First, find one tetrahedron whose open circumball contains *u* (point location). Second, a depth-first search in the triangulation finds all the other tetrahedra whose open circumballs contain *u*, in time proportional to their number. Third, delete these tetrahedra, as illustrated in Figure 5.2. The union of the deleted tetrahedra is a star-shaped polyhedral cavity. Fourth, for each triangular face of the cavity, create a new tetrahedron joining it with *u*, as illustrated.

To support inserting vertices that lie outside the triangulation, each triangular face on the boundary of the triangulation adjoins a *ghost tetrahedron* analogous to the ghost triangles of Section 3.4, having three solid vertices and a ghost vertex *g*. A tetrahedron that is not a ghost is called *solid*. Let *vwx* be a boundary triangle, oriented so its back adjoins a positively oriented solid tetrahedron *xwvy*. The incremental insertion algorithm stores a positively oriented ghost

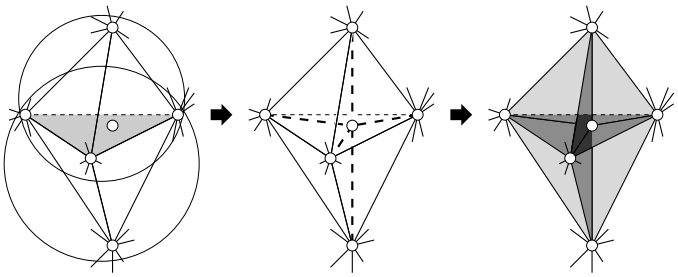

Figure 5.2: The Bowyer–Watson algorithm in three dimensions. A new vertex falls in the open circumballs of the two tetrahedra illustrated at left. These tetrahedra may be surrounded by other tetrahedra, which for clarity are not shown. The two tetrahedra and the face they share (shaded) are deleted. At center, the five new Delaunay edges. At right, the nine new Delaunay triangles—one for each edge of the cavity. Six new tetrahedra are created—one for each facet of the cavity.

tetrahedron *vwxg* in the triangulation data structure.

When a new vertex is inserted, there are two cases in which *vwxg* must be deleted, i.e. *vwx* is no longer a boundary triangle: if a vertex is inserted in the open halfspace in front of *vwx*, or if a newly inserted vertex lies in the open circumdisk of *vwx* (i.e. it is coplanar with *vwx* and in its open diametric ball). Call the union of these two regions the *outer halfspace* of *vwx*. It is the set of points in the open circumball of *vwxg* in the limit as *g* moves away from the triangulation.

The following pseudocode details the Bowyer–Watson algorithm in three dimensions, omitting point location. The parameters to I V 3D are a vertex *u* to insert and a positively oriented tetrahedron *vwxy* whose open circumball contains *u*. In this pseudocode, all the triangles and tetrahedra are oriented, so the vertex order matters.

I V 3D($u, vwxy$)

1. Call D T (v, w, x, y).
2. Call D C 3D(u, xwv), D C 3D(u, yvw), D C 3D(u, vyx), and D - C 3D(u, wxy) to identify the other deleted tetrahedra and insert new tetrahedra.

D C 3D(u, wxy)

3. Let z = A (w, x, y); *wxyz* is the tetrahedron on the other side of facet *wxy* from *u*.
4. If $z = \emptyset$, then return, because the tetrahedron has already been deleted.
5. If I S $(u, w, x, y, z) > 0$, then *uwxy* and *wxyz* are not Delaunay, so call D T (w, x, y, z). Call D C 3D(u, wxz), D C 3D(u, xyz), and D C 3D(u, ywz) to recursively identify more deleted tetrahedra and insert new tetrahedra, and return.
6. Otherwise, *wxy* is a face of the polyhedral cavity, so call A T (u, w, x, y).

The correctness of I V 3D depends on the use of a ghost vertex. In particular, step 4 should not confuse the ghost vertex with ∅; the former marks the triangulation exterior and the latter marks the cavity. Unlike with the planar algorithm I V in Section 3.4, Step 4 is necessary for both unweighted and weighted Delaunay triangulations.

Step 5 requires a modification to the I S test discussed in Section 3.1: if *wxyz* is a ghost tetrahedron, then replace the formula (3.5) with a test of whether *u* lies in the outer half-space of *wxyz*. To adapt the code for a weighted Delaunay triangulation, replace (3.5) with O 4D(u^+, w^+, x^+, y^+, z^+), which tests whether u^+ is below the witness plane of *wxyz*. The parameter *vwxy* of I V 3D must be a tetrahedron whose witness plane is above u^+.

How expensive is vertex insertion, leaving out the cost of point location? The insertion of a single vertex into an *n*-vertex Delaunay triangulation can delete $\Theta(n^2)$ tetrahedra if the triangulation is the one depicted in Figure 4.2. However, a single vertex insertion can only create $\Theta(n)$ tetrahedra: observe that the boundary of the cavity is a planar graph, so the cavity has fewer than $2n$ boundary triangles.

It follows that during a sequence of *n* vertex insertion operations, at most $\Theta(n^2)$ tetrahedra are created. A tetrahedron can only be deleted if it is first created, so at most $\Theta(n^2)$ tetrahedra are deleted, albeit possibly most of them in a single vertex insertion. For the worst points sets, randomizing the vertex insertion order does not improve these numbers.

A special case that occurs frequently in practice—by all accounts it seems to be the norm—is the circumstance where the Delaunay triangulation has complexity linear, rather than quadratic, in the number of vertices, and moreover the intermediate triangulations produced during incremental insertion have expected linear complexity. For point sets with this property, a random insertion order guarantees that each vertex insertion will create and delete an expected constant number of tetrahedra, just as it does in the plane, and we shall see that the randomized incremental insertion algorithm with a conflict graph runs in expected $O(n \log n)$ time. This running time is often observed in practice, even in higher dimensions. Be forewarned, however, that there are point sets for which the final triangulation has linear complexity but the intermediate triangulations have expected quadratic complexity, thereby slowing down the algorithm dramatically.

Even for worst-case point sets, randomization helps to support fast point location. Recall that, excluding the point location step, the Bowyer–Watson algorithm runs in time proportional to the number of tetrahedra it deletes and creates, so the running time of the three-dimensional incremental insertion algorithm, *excluding* point location, is $O(n^2)$. With a conflict graph and a random insertion order, point location is no more expensive than this, so the randomized incremental insertion algorithm achieves a worst-case optimal expected running time of $O(n^2)$.

5.3 Biased randomized insertion orders

The advantage of inserting vertices in random order is that it guarantees that the expected running time of point location is optimal, and that pathologically slow circumstances like those illustrated in Figure 3.8 are unlikely to happen. But there is a serious disadvantage: randomized vertex insertions tend to interact poorly with the memory hierarchy in modern computers, especially virtual memory. Ideally, data structures representing tetrahedra and vertices that are close together geometrically should be close together in memory—a property called *spatial locality*—for better cache and virtual memory performance.

Fortunately, the permutation of vertices does not need to be uniformly random for the running time to be asymptotically optimal. A *biased randomized insertion order* (BRIO) is a permutation of the vertices that has strong spatial locality but retains enough randomness to obtain an expected running time of $O(n^2)$. Experiments show that a BRIO greatly improves the efficiency of the memory hierarchy—especially virtual memory.

Experiments also show that incremental insertion achieves superior running times in practice when it uses a BRIO but replaces the conflict graph with a point location method that simply walks from the previously inserted vertex toward the next inserted vertex; see Section 5.5. Although walking point location does not offer a strong theoretical guarantee on running time like a conflict graph does, this incremental insertion algorithm is perhaps the most attractive in practice, as it combines excellent observed speed with a simple implementation.

Let n be the number of vertices to triangulate. A BRIO orders the vertices in a sequence of *rounds* numbered zero through $\lceil \log_2 n \rceil$. Each vertex is assigned to the final round, round $\lceil \log_2 n \rceil$, with probability $1/2$. The remaining vertices are assigned to the second-last round with probability $1/2$, and so on. Each vertex is assign to round zero with probability $(1/2)^{\lceil \log_2 n \rceil} \leq 1/n$. The incremental insertion algorithm begins by inserting the vertices in round zero, then round one, and so on to round $\lceil \log_2 n \rceil$.

Within any single round, the vertices can be arranged in any order without threatening the worst-case expected running time of the algorithm, as Section 5.4 proves. Hence, we order the vertices within each round to create as much spatial locality as possible. One way to do this is to insert the vertices in the order they are encountered on a space-filling curve such as a Hilbert curve or a z-order curve. Another way is to store the vertices in an octree or k-d tree, refined so each leaf node contains only a few vertices; then order the vertices by a traversal of the tree. (Octree traversal is one way to sort vertices along a Hilbert or z-order curve.)

The tendency of vertices that are geometrically close together to be close together in the ordering does not necessarily guarantee that the data structures associated with them will be close together in memory. Nevertheless, experiments show that several popular Delaunay triangulation programs run faster with a BRIO than with a vertex permutation chosen uniformly at random, especially when the programs run out of main memory and have to resort to virtual memory.

Whether one uses the traditional randomized incremental insertion algorithm or a BRIO, one faces the problem of bootstrapping the algorithm, as discussed in Section 3.7. The most practical approach is to choose four affinely independent vertices, construct their Delaunay triangulation (a single tetrahedron), create four adjoining ghost tetrahedra, construct a conflict graph, and insert the remaining vertices in a random order (a uniformly chosen permutation or a BRIO). Even if the four bootstrap vertices are not chosen randomly, it is possible to prove that the expected asymptotic running time of the algorithm is not compromised.

5.4 Optimal point location by a conflict graph in $\mathbb{R}^3$

Section 3.6 describes how to use a conflict graph to perform point location. Conflict graphs generalize to higher dimensions, yielding a randomized incremental insertion algorithm that constructs three-dimensional Delaunay triangulations in expected $O(n^2)$ time, which is optimal in the worst case. Moreover, the algorithm runs in expected $O(n \log n)$ time in the special case where the Delaunay triangulation of a random subset of the input points has expected linear complexity.

Conflict graphs and the vertex redistribution algorithm extend to three or more dimensions straightforwardly, with no new ideas needed. A conflict is a vertex-tetrahedron pair consisting of an uninserted vertex and a tetrahedron whose open circumball contains it. For each uninserted vertex, the conflict graph records a tetrahedron that contains the vertex. If an uninserted vertex lies outside the growing triangulation, its conflict is an unbounded, convex ghost "tetrahedron," each having one solid triangular facet on the triangulation boundary and three unbounded ghost facets. The ghost edges and ghost facets diverge from a point in the interior of the triangulation (recall Figure 3.10). For each tetrahedron, the conflict graph records a list of the uninserted vertices that choose that tetrahedron as their conflict. When a vertex is inserted into the triangulation, the conflict graph is updated exactly as described in Section 3.6.

Let us analyze the running time. No backward analysis is known for a BRIO, so the analysis given here differs substantially from that of Section 3.6. This analysis requires us to assume that the point set is generic, although the algorithm is just as fast for point sets that are not.

Let S be a generic set of n points in $\mathbb{R}^3$, not all coplanar. Let σ be a tetrahedron whose vertices are in S. We call σ a *j-tetrahedron* if its open circumball contains j vertices in S.

Because S is generic, $\mathrm{Del}\, S$ is composed of all the 0-tetrahedra of S. However, the incremental insertion algorithm transiently constructs many other tetrahedra that do not survive to the end; these are 1-tetrahedra, 2-tetrahedra, and so forth. We wish to determine, for each j, how many j-tetrahedra exist and what is the probability that any one of them is constructed. From this we can determine the expected number of j-tetrahedra that appear.

The *triggers* of a j-tetrahedron are its four vertices, and the *stoppers* of a j-tetrahedron are the j vertices its open circumball contains. A tetrahedron is constructed if and only if all of its triggers precede all of its stoppers in the insertion order. The probability of that decreases rapidly as j increases.

Let f_j be the total number of j-tetrahedra, which depends on S, and let $F_j = \sum_{i=0}^{j} f_i$ be the total number of i-tetrahedra for all $i \leq j$. $\mathrm{Del}\, S$ contains at most $O(n^2)$ tetrahedra, so $F_0 = f_0 = O(n^2)$. Unfortunately, it is a difficult open problem to find a tight bound on the number f_j; we must settle for a tight bound on the number F_j. The following proposition is an interesting use of the probabilistic method. It exploits the fact that if we compute the Delaunay triangulation of a random subset of S, we know the probability that any given j-tetrahedron will appear in the triangulation.

Proposition 5.1. *For a generic set S of n points, $F_j = O(j^2 n^2)$. If the Delaunay triangulation of a random r-point subset of S has expected $O(r)$ complexity for every $r < n$, then $F_j = O(j^3 n)$.*

P . Let R be a random subset of S, where each point in S is chosen with probability $1/j$. (This is a magical choice, best understood by noting that for any j-tetrahedron, the expected number of its stoppers in R is one.) Let $r = |R|$. Observe that r is a random variable with a binomial distribution. Therefore, the expected complexity of $\mathrm{Del}\, R$ is $O(E[r^2]) = O(E[r]^2) = O(n^2/j^2)$.

Let σ be a k-tetrahedron for an arbitrary k. Because S is generic, $\mathrm{Del}\, R$ contains σ if and only if R contains all four of σ's triggers but none of its k stoppers. The probability of that is

$$p_k = \left(\frac{1}{j}\right)^4 \left(1 - \frac{1}{j}\right)^k .$$

If $k \leq j$, then

$$p_k \geq \left(\frac{1}{j}\right)^4 \left(1 - \frac{1}{j}\right)^j \geq \left(\frac{1}{j}\right)^4 \frac{1}{e},$$

where e is the base of the natural logarithm. This inequality follows from the identity $\lim_{j\to\infty}(1 + x/j)^j = e^x$.

The expected number of tetrahedra in $\operatorname{Del} R$ is

$$E[\text{size}(\operatorname{Del} R)] = \sum_{k=0}^{n} p_k f_k \geq \sum_{k=0}^{j} p_k f_k \geq \sum_{k=0}^{j} \left(\frac{1}{j}\right)^4 \frac{f_k}{e} = \frac{F_j}{e j^4}.$$

Recall that this quantity is $O(n^2/j^2)$. Therefore,

$$F_j \leq e j^4 E[\text{size}(\operatorname{Del} R)] = O(j^2 n^2).$$

If the expected complexity of the Delaunay triangulation of a random subset of S is linear, the expected complexity of $\operatorname{Del} R$ is $O(E[r]) = O(n/j)$, so $F_j = O(j^3 n)$. □

Proposition 5.1 places an upper bound on the number of j-tetrahedra with j small. Next, we wish to know the probability that any particular j-tetrahedron will be constructed during a run of the randomized incremental insertion algorithm.

Proposition 5.2. *If the permutation of vertices is chosen uniformly at random, then the probability that a specified j-tetrahedron is constructed during the randomized incremental insertion algorithm is less than* $4!/j^4$.

P . A j-tetrahedron appears only if its four triggers (vertices) appear in the permutation before its j stoppers. This is the probability that if four vertices are chosen randomly from the $j + 4$ triggers and stoppers, they will all be triggers; namely

$$\frac{1}{\binom{j+4}{4}} = \frac{j!4!}{(j+4)!} < \frac{4!}{j^4}.$$

□

Proposition 5.2 helps to bound the expected running time of the standard randomized incremental insertion algorithm, but we need a stronger proposition to bound the running time with a biased randomized insertion order.

Proposition 5.3. *If the permutation of vertices is a BRIO, as described in Section 5.3, then the probability q_j that a specified j-tetrahedron will be constructed during the incremental insertion algorithm is less than*

$$\frac{16 \cdot 4! + 1}{j^4}.$$

P . Let σ be a j-tetrahedron. Let i be the round in which its first stopper is inserted. The incremental insertion algorithm can create σ only if its last trigger is inserted in round i or earlier.

Consider a trigger that is inserted in round i or earlier for any $i \neq 0$. The probability of that trigger being inserted before round i is $1/2$. Therefore, the probability that all four triggers are inserted in round $i \neq 0$ or earlier is exactly 2^4 times greater than the probability that all four triggers are inserted before round i. Therefore, the probability q_j that the algorithm creates σ is bounded as follows.

$$\begin{aligned}
q_j &\leq \text{Prob[last trigger round} \leq \text{first stopper round]} \\
&\leq 2^4 \text{ Prob[last trigger round} < \text{first stopper round]} + \text{Prob[last trigger round} = 0] \\
&\leq 16 \text{ Prob[all triggers appear before all stoppers in} \\
&\qquad\qquad \text{a uniform random permutation]} + (1/2)^{4\lceil \log_2 n \rceil} \\
&< 16\frac{4!}{j^4} + \frac{1}{n^4} \\
&\leq \frac{16 \cdot 4! + 1}{j^4}.
\end{aligned}$$

The fourth line of this inequality follows from Proposition 5.2. □

Theorem 5.4. *The expected running time of the randomized incremental insertion algorithm on a generic point set S in three dimensions, whether with a uniformly random permutation or a BRIO, is $O(n^2)$. If the Delaunay triangulation of a random r-point subset of S has expected $O(r)$ complexity for every $r \leq n$, then the expected running time is $O(n \log n)$.*

P . Section 3.6 shows that the total running time of the algorithm is proportional to the number of tetrahedra created plus the number of conflicts created. The expected number of tetrahedra created is $\sum_{j=0}^{n} q_j f_j$, with q_j defined as in Proposition 5.3. Because a j-tetrahedron participates in j conflicts, the expected number of conflicts created is $\sum_{j=0}^{n} j q_j f_j$. Hence, the expected running time is

$$O\left(\sum_{j=0}^{n} q_j f_j + \sum_{j=0}^{n} j q_j f_j\right) \leq O\left(n^2 + \sum_{j=1}^{n} j q_j f_j\right).$$

The $O(n^2)$ term accounts for the first term of the first summation, for which $j = 0$ and $q_j = 1$, because the algorithm constructs every 0-tetrahedron. The second summation is bounded as

follows.

$$\begin{aligned}
\sum_{j=1}^{n} jq_j f_j &< \sum_{j=1}^{n} \frac{16 \cdot 4! + 1}{j^3}(F_j - F_{j-1}) \\
&= (16 \cdot 4! + 1)\left(\sum_{j=1}^{n} \frac{F_j}{j^3} - \sum_{k=1}^{n} \frac{F_{k-1}}{k^3}\right) \\
&= (16 \cdot 4! + 1)\left(\sum_{j=1}^{n-1} \frac{F_j}{j^3} + \frac{F_n}{n^3} - \sum_{k=2}^{n} \frac{F_{k-1}}{k^3} - F_0\right) \\
&= (16 \cdot 4! + 1)\left(\sum_{j=1}^{n-1} \left(\frac{F_j}{j^3} - \frac{F_j}{(j+1)^3}\right) + \frac{F_n}{n^3} - F_0\right) \\
&< (16 \cdot 4! + 1)\left(\sum_{j=1}^{n-1} \frac{F_j}{j^3(j+1)^3}\left((j+1)^3 - j^3\right) + \frac{F_n}{n^3}\right) \\
&= O\left(\sum_{j=1}^{n-1} \frac{j^2 n^2}{j^6} j^2 + n\right) \\
&= O\left(n^2 \sum_{j=1}^{n-1} \frac{1}{j^2} + n\right) \\
&= O(n^2).
\end{aligned}$$

The last line follows from Euler's identity $\sum_{j=1}^{\infty} 1/j^2 = \pi^2/6$. If the expected complexity of the Delaunay triangulation of a random subset of S is linear, $F_j = O(j^3 n)$, so the expected running time is $O(n \sum_{j=1}^{n-1}(1/j)) = O(n \log n)$. □

5.5 Point location by walking

In conjunction with a BRIO, a simple point location method called *walking* appears to outperform conflict graphs in practice, although there is no guarantee of a fast running time. A walking point location algorithm simply traces a straight line through the triangulation, visiting tetrahedra that intersect the line as illustrated in Figure 5.3, until it arrives at a tetrahedron that contains the new vertex. In conjunction with a vertex permutation chosen uniformly at random (rather than a BRIO), walking point location visits many tetrahedra and is very slow. But walking is fast in practice if it follows two guidelines: the vertices should be inserted in an order that has much spatial locality, such as a BRIO, and each walk should begin at the most recently created solid tetrahedron. Then the typical walk visits a small constant number of tetrahedra.

To avoid a long walk between rounds of a BRIO, the vertex order (e.g. the tree traversal or the direction of the space-filling curve) should be reversed on even-numbered rounds, so each round begins near where the previous round ends.

Researchers have observed that the three-dimensional incremental insertion algorithm with a BRIO and walking point location appears to run in linear time, not counting the initial $O(n \log n)$-

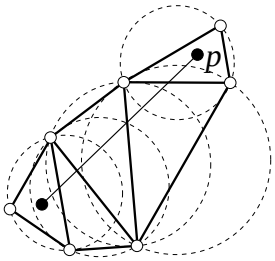

Figure 5.3: Walking to the triangle that contains p.

time computation of a BRIO. This observation holds whether they use a BRIO or a spatial ordering generated by traversing an octree with no randomness at all. Randomness is often unnecessary in practice—frequently, simply sorting the vertices along a space-filling curve will yield excellent speed—but because points sets like that illustrated in Figure 3.8 are common in practice, we recommend choosing a BRIO to prevent the possibility of a pathologically slow running time.

5.6 The gift-wrapping algorithm in $\mathbb{R}^3$

The simplest algorithm for retriangulating the cavity evacuated when a vertex is deleted from a three-dimensional Delaunay triangulation or CDT, or when a polygon is inserted or deleted in a CDT, is gift-wrapping. (See the bibliographic notes for more sophisticated vertex deletion algorithms, also based on gift-wrapping, that are asymptotically faster in theory.) The gift-wrapping algorithm described in Section 3.11 requires few new ideas to work in three (or more) dimensions. The algorithm constructs tetrahedra one at a time, and maintains a dictionary of unfinished triangular facets. The pseudocode for F and G W CDT can be adapted, with triangles replaced by tetrahedra, oriented edges replaced by oriented facets, and circumdisks replaced by circumballs.

The biggest change is that triangles, not segments, seed the algorithm. But the polygons in a PLC are not always triangles. Recall from Proposition 4.11 that a CDT of a PLC $\mathcal{P}$ induces a two-dimensional CDT of each polygon in $\mathcal{P}$. To seed the three-dimensional gift-wrapping algorithm, one can compute the two-dimensional CDT of a polygon (or every polygon), then enter each CDT triangle (twice, with both orientations) in the dictionary.

To gift-wrap a Delaunay triangulation, seed the algorithm with one strongly Delaunay triangle. One way to find one is to choose an arbitrary input point and its nearest neighbor. For the third vertex of the triangle, choose the input point that minimizes the radius of the circle through the three vertices. If the set of input points is generic, the triangle having these three vertices is strongly Delaunay.

If the input (point set or PLC) is not generic, gift-wrapping is in even greater danger in three dimensions than in the plane. Whereas the planar gift-wrapping algorithm can handle subsets of four or more cocircular points by identifying them and giving them special treatment, no such approach works reliably in three dimensions. Imagine a point set that includes six points lying on a common empty sphere. Suppose that gift-wrapping inadvertently tetrahedralizes the space around these points so they are the vertices of a hollow cavity shaped like Schönhardt's polyhedron (from Section 4.5). The algorithm will be unable to fill the cavity. By far the most practical solution is

to symbolically perturb the points so that they are generic, as discussed in Section 2.9. The same perturbation should also be used to compute the two-dimensional CDTs of the PLC's polygons.

Another difficulty is that the input PLC might not have a CDT, in which case gift-wrapping will fail in one of two ways. One possibility is that the algorithm will fail to finish an unfinished facet, even though there is a vertex in front of that facet, because no vertex in front of that facet is visible from the facet's interior. This failure is easy to detect. The second possibility is that the algorithm will finish a facet by constructing a tetrahedron that is not constrained Delaunay, either because the tetrahedron's open circumball contains a visible vertex, or because the tetrahedron intersects the preexisting simplices wrongly (not in a complex). An attempt to gift-wrap Schönhardt's polyhedron brings about the last fate. The algorithm becomes substantially slower if it tries to detect these failures. Perhaps a better solution is to run the algorithm only on PLCs that are edge-protected or otherwise known to have CDTs.

A strange property of the CDT is that it is NP-hard to determine whether a three-dimensional PLC has a CDT, if the PLC is not generic. However, a polynomial-time algorithm is available for generic PLCs: run the gift-wrapping algorithm, and check whether it succeeded.

Gift-wrapping takes $O(nh)$ time for a Delaunay triangulation, or $O(nmh)$ time for a CDT, where n is the number of input points, m is the total complexity of the input polygons, and h is the number of tetrahedra in the CDT; h is usually linear in n, but could be quadratic in the worst case.

5.7 Inserting a vertex into a CDT in $\mathbb{R}^3$

Section 3.9 describes how to adapt the Bowyer–Watson vertex insertion algorithm to CDTs in the plane. The same adaptions work for three-dimensional CDTs, but there is a catch: even if a PLC $\mathcal{P}$ has a CDT, an augmented PLC $\mathcal{P} \cup \{v\}$ might not have one. This circumstance can be diagnosed after the depth-first search step of the Bowyer–Watson algorithm in one of two ways: by the fact that the cavity is not star-shaped, thus one of the newly created tetrahedra has nonpositive orientation, or by the fact that a segment or polygon runs through the interior of the cavity. An implementation can check explicitly for these circumstances, and signal that the vertex v cannot be inserted.

5.8 Inserting a polygon into a CDT

To "insert a polygon into a CDT" is to take as input the CDT $\mathcal{T}$ of some PLC $\mathcal{P}$ and a new polygon f to insert, and produce the CDT of $\mathcal{P}^f = \mathcal{P} \cup \{f\}$. It is only meaningful if $\mathcal{P}^f$ is a valid PLC—which implies that f's boundary is a union of segments in $\mathcal{P}$, among other things. It is only possible if $\mathcal{P}^f$ has a CDT. If $\mathcal{P}$ is edge-protected, then $\mathcal{P}^f$ is edge-protected (polygons play no role in the definition of "edge-protected"), and both have CDTs. But if $\mathcal{P}$ is not edge-protected, it is possible that $\mathcal{P}$ has a CDT and $\mathcal{P}^f$ does not; see Exercise 5.

The obvious algorithm echoes the segment insertion algorithm in Section 3.10: delete the tetrahedra whose interiors intersect f. All the simplices not deleted are still constrained Delaunay. Then retriangulate the polyhedral cavities on either side of f with constrained Delaunay simplices, perhaps by gift-wrapping. (Recall Figure 2.15.) Note that if $\mathcal{P}^f$ has no CDT, the retriangulation step will fail.

This section describes an alternative algorithm that uses bistellar flips to achieve the same result. One can construct a three-dimensional CDT of an n-vertex, ridge-protected PLC $\mathcal{P}$ in $O(n^2 \log n)$ time by first constructing a Delaunay triangulation of the vertices in $\mathcal{P}$ in expected $O(n^2)$ time with the randomized incremental insertion algorithm, then inserting the polygons one by one with the flip-based algorithm. For most PLCs that arise in practice, this CDT construction algorithm is likely to run in $O(n \log n)$ time and much faster than gift-wrapping.

The algorithm exploits the fact that when the vertices of the triangulation are lifted by the parabolic lifting map, every locally Delaunay facet lifts to a triangle where the lifted triangulation is locally convex. Say that a facet g, shared by two tetrahedra σ and τ, is *locally weighted Delaunay* if the lifted tetrahedra σ^+ and τ^+ adjoin each other at a dihedral angle, measured from above, of less than 180°. In other words, the interior of τ^+ lies above the affine hull of σ^+, and vice versa.

The algorithm's main idea is to move some of the lifted vertices vertically, continuously, and linearly so they rise above the paraboloid, and use bistellar flips to dynamically maintain local convexity as they rise. Recall from Figure 4.7 that a tetrahedral bistellar flip is a transition between the upper and lower faces of a 4-simplex. If the vertices of that simplex are moving vertically at different (but constant) speeds, they may pass through an instantaneous state in which the five vertices of the 4-simplex are cohyperplanar, whereupon the lower and upper faces are exchanged. In the facet insertion algorithm, this circumstance occurs when two lifted tetrahedra σ^+ and τ^+ that share a triangular facet g^+ become cohyperplanar, whereupon the algorithm uses the procedure F from Section 4.4 to perform a bistellar flip that deletes g.

The algorithm for inserting a polygon f into a triangulation $\mathcal{T}$ begins by identifying a region R: the union of the tetrahedra in $\mathcal{T}$ whose interiors intersect f, and thus must be deleted. The polygon insertion algorithm only performs flips in the region R. Let h be the plane aff f. Call the vertices in $\mathcal{P}$ on one (arbitrary) side of h *left vertices*, and the vertices on the other side *right vertices*. Vertices on h are neither. The flip algorithm linearly increases the heights of the vertices according to their distance from h, and uses flips to maintain locally weighted Delaunay facets in R as the heights change. Figure 5.4 is a sequence of snapshots of the algorithm at work.

Assign each vertex $v \in \mathcal{P}$ a time-varying *height* of $v_z(\kappa) = \|v\|^2 + \kappa d(v, h)$, where κ is the time and $d(v, h)$ is the Euclidean distance of v from h. (This choice of $d(\cdot, \cdot)$ is pedagogically useful but numerically poor; a better choice for implementation is to let $d(v, h)$ be the distance of v from h along one coordinate axis, preferably the axis most nearly perpendicular to h. This distance is directly proportional to the Euclidean distance, but can be computed without radicals.)

When a set of vertices is transformed affinely, its convex hull undergoes no combinatorial change. Likewise, an affine transformation of the vertex heights in a lifted triangulation does not change which facets are locally weighted Delaunay. In the facet insertion algorithm, however, each half of space undergoes a different affine transformation, so the simplices that cross the plane h change as the time κ increases. Observe that an algorithm in which only the heights of the right vertices change (at twice the speed) is equivalent. For numerical reasons, it is better to raise only half the vertices.

Let $\mathcal{P}(\kappa)$ be a time-varying weighted PLC, which is identical to $\mathcal{P}$ except that each right vertex v is lifted to a height of $v_z(\kappa)$. As κ increases, the algorithm F I P maintains a triangulation of R that is *weighted constrained Delaunay* with respect to $\mathcal{P}(\kappa)$, meaning that every triangular facet is locally weighted Delaunay except those included in a polygon.

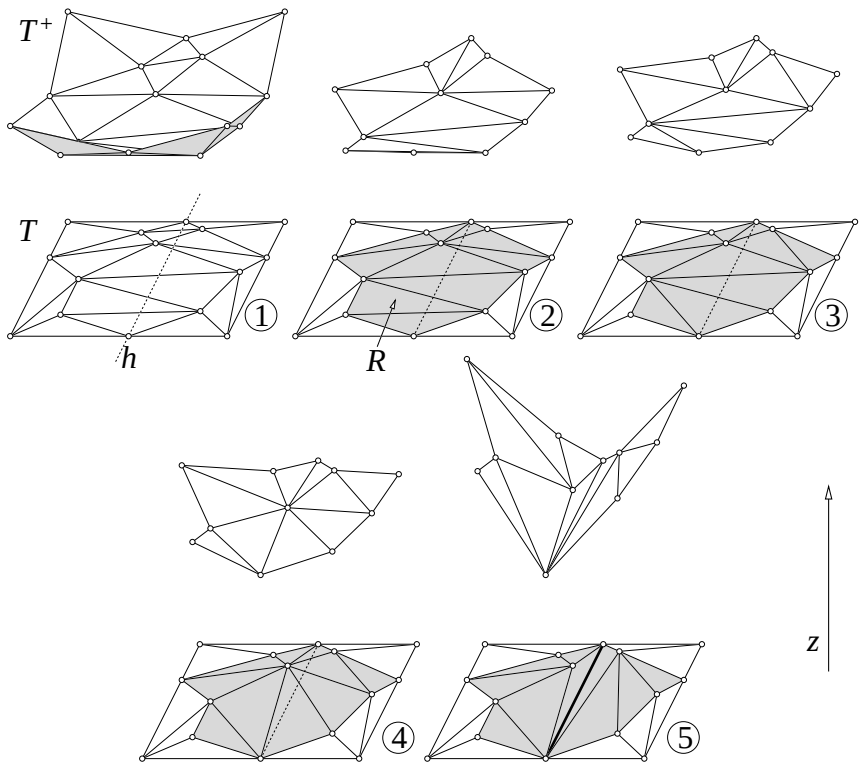

Figure 5.4: A two-dimensional example of inserting a segment into a CDT. The algorithm extends to any dimension.

Every simplex in the evolving triangulation that has no left vertex, or no right vertex, remains constrained Delaunay with respect to $\mathcal{P}(\kappa)$ as κ increases. The algorithm F I P deletes only simplices that have both a left and a right vertex *and* pass through f. All simplices outside the region R, or strictly on the boundary of R, remain intact.

When κ is sufficiently large, the flip algorithm reaches a state where no simplex in the region R has both a left vertex and a right vertex, hence f is a union of faces of the triangulation. At this time, the triangulation is the CDT of $\mathcal{P}^f$, and the job is done.

Pseudocode for F I P appears below. The loop (step 4) dynamically maintains the triangulation $\mathcal{T}$ as κ increases from 0 to ∞ and the lifted companions of the right vertices move up. For certain values of κ, the following event occurs: some facet g in the region R is no longer locally weighted Delaunay after time κ, because the two lifted tetrahedra that include g^+ are cohyperplanar at time κ. Upon this event, an update operation replaces these and other simplices that will not be locally weighted Delaunay after time κ with simplices that will be.

To ensure that it performs each bistellar flip at the right time, the algorithm maintains a priority queue (e.g. a binary heap) that stores any flip that might occur. For each facet g that could be flipped at some time in the future, the procedure C determines when g might be flipped and enqueues a flip event. The main loop of F I P repeatedly removes the flip with the least time from the priority queue, and performs a flip if the facet still exists, i.e. was not eliminated by other flips. When the queue is empty, $\mathcal{T}$ has transformed into the CDT of $\mathcal{P}^f$.

F I P ($\mathcal{T}, f$)

1. Find one tetrahedron in $\mathcal{T}$ that intersects the interior of f by a rotary search around an edge of f.
2. Find all tetrahedra in $\mathcal{T}$ that intersect the interior of f by a depth-first search.
3. Initialize Q to be an empty priority queue. For each facet g in $\mathcal{T}$ that intersects the interior of f and has a vertex on each side of f, call C (g, Q).
4. While Q is not empty
   (a) Remove (g', κ) with minimum κ from Q.
   (b) If g' is still a facet in $\mathcal{T}$:
      (i) Call F ($\mathcal{T}, g'$) to eliminate g'.
      (i) For each facet g that lies on the boundary of the cavity retriangulated by the flip and has a vertex on each side of f, call C (g, Q).

C (g, Q)

1. Let σ and τ be the tetrahedra that share the facet g.
2. If the interior of σ^+ will be below aff τ^+ at time ∞:
   (a) Compute the time κ at which σ^+ and τ^+ are cohyperplanar.
   (b) Insert (g, κ) into the priority queue Q.

Just as gift-wrapping can fail when $\mathcal{P}$ is not generic, F F P can fail when $\mathcal{P}$ is not generic or simultaneous events occur for other reasons. For an implementation to succeed in practice, step 2 of C should perturb the vertex weights as described in Section 2.9 and in Exercise 2 of Chapter 3. It is possible for several events that take place at the same time (or nearly the same time, if roundoff error occurs) to come off the priority queue in an unexecutable order—recall that there are several circumstances annotated in the F pseudocode in which it might be impossible to perform a flip that eliminates a specified facet. In these circumstances, step 4(a) of F I P should dequeue an event with an admissible flip and hold back events with inadmissible flips until they become admissible.

The correctness proof for F I P is omitted, but it relies on the Constrained Delaunay Lemma. All the tetrahedra outside the region R remain constrained Delaunay, so their facets are constrained Delaunay, except those included in a polygon. When the algorithm is done, the original vertex heights are restored (returning them to the paraboloid). This is an affine transformation of the lifted right vertices, so the facets created by bistellar flips remain locally Delaunay, except the facets included in f. Therefore, every facet not included in a polygon is locally Delaunay. By the Constrained Delaunay Lemma, the final triangulation is a CDT of $\mathcal{P}^f$.

For an analysis of the running time, let m be the number of vertices in the region R, and let n be the number of vertices in $\mathcal{T}$.

Proposition 5.5. *Steps 3 and 4 of* F I P *run in* $O(m^2 \log m)$ *time.*

P . Every bistellar flip either deletes or creates an edge in the region R. Because the vertex weights vary linearly with time, an edge that loses the weighted constrained Delaunay property

will never regain it; so once an edge is deleted, it is never created again. Therefore, FIP deletes fewer than $m(m-1)/2$ edges and creates fewer than $m(m-1)/2$ edges over its lifetime, and thus performs fewer than m^2 flips. Each flip enqueues at most a constant number of events. Each event costs $O(\log m)$ time to enqueue and dequeue, yielding a total cost of $O(m^2 \log m)$ time. □

Miraculously, the worst-case running time for any sequence of FIP operations is only a constant factor larger than the worst-case time for one operation.

Proposition 5.6. *The running time of any sequence of valid* FIP *operations applied consecutively to a triangulation is* $O(n^2 \log n)$*, if step 1 of* FIP *is implemented efficiently enough.*

P. A sequence of calls to FIP, like a single call, has the property that every simplex deleted is never created again. (Every deleted simplex crosses a polygon, and cannot return after the polygon is inserted.) Therefore, a sequence of calls deletes fewer than $n(n-1)/2$ edges, creates fewer than $n(n-1)/2$ edges, and performs fewer than n^2 flips. Each flip enqueues a constant number of events. Each event costs $O(\log n)$ time to enqueue and dequeue, summing to $O(n^2 \log n)$ time for all events.

The cost of step 2 of FIP (the depth-first search) is proportional to the number of tetrahedra that intersect the relative interior of the polygon f. These tetrahedra either are deleted when f is inserted, or they intersect f's relative interior without crossing f. A tetrahedron can intersect the relative interiors of at most ten PLC polygons that it does not cross, so each tetrahedron is visited in at most eleven depth-first searches. At most $O(n^2)$ tetrahedra are deleted and created during a sequence of calls to FIP. Therefore, the total cost of step 2 over all calls to FIP is $O(n^2)$.

The cost of step 1 of FIP (identifying a tetrahedron that intersects the relative interior of f) is $O(n)$. This is a pessimistic running time; the worst case is achieved only if the edge used for the rotary search is an edge of $\Theta(n)$ facets. If $\Omega(n \log n)$ polygons are inserted into a single triangulation (which is possible but unlikely in practice), we must reduce the cost of step 1 below $O(n)$. This can be done by giving each segment a balanced search tree listing the adjoining facets in the triangulation, in rotary order around the segment. Then, step 1 executes in $O(\log n)$ time. The balanced trees are updated in $O(\log n)$ time per facet created or deleted. □

Recall from Section 4.4 that a bistellar flip replaces the top faces of a 4-simplex with its bottom faces. Because no face reappears after it is deleted, the sequence of flips performed during incremental polygon insertion is structurally similar to a four-dimensional triangulation. It has often been observed that most practical point sets have linear-size Delaunay triangulations in three or higher dimensions, so it seems like a reasonable inference that for most practical PLCs, the sequence of flips should have linear length. For those PLCs, incremental CDT construction with FIP runs in $\Theta(n \log n)$ time.

5.9 Notes and exercises

An incremental insertion algorithm that works in any dimension was discovered independently by Bowyer [33], Hermeline [111, 112], and Watson [222]. Bowyer and Watson submitted their articles to *Computer Journal* and found them published side by side in 1981.

Although there is rarely a reason to choose them over the Bowyer–Watson algorithm, there is a literature on vertex insertion algorithms that use bistellar flips. Joe [118] generalizes Lawson's flip-based vertex insertion algorithm [131] to three dimensions, Rajan [173] generalizes it to higher dimensions, Joe [119] improves the speed of Rajan's generalization, and Edelsbrunner and Shah [91] generalize Joe's latter algorithm to weighted Delaunay triangulations.

Clarkson and Shor [64] introduce conflict graphs and show that randomized incremental insertion with a conflict graph runs in expected $O(n^{\lceil d/2 \rceil})$ time for Delaunay triangulations in $d \geq 3$ dimensions. Amenta, Choi, and Rote [7] propose the idea of a biased randomized insertion order and give the analysis of the randomized incremental insertion algorithm in Section 5.4. The bound on the number of j-tetrahedra in Proposition 5.1 was itself a major breakthrough of Clarkson and Shor [64]. The simpler proof given here is due to Mulmuley [154]. See Seidel [194] for a backward analysis of the algorithm with a vertex permutation chosen uniformly at random, in the style of Section 3.6.

Guibas and Stolfi [106] give an algorithm for walking point location in a planar Delaunay triangulation, and Devillers, Pion, and Teillaud [72] compare walking point location algorithms in two and three dimensions. The discussion in Section 5.5 of combining a BRIO with walking point location relies on experiments reported by Amenta, Choi, and Rote [7].

The first gift-wrapping algorithm for constructing Delaunay triangulations in three or more dimensions appeared in 1979 when Brown [34] published his lifting map and observed that the 1970 gift-wrapping algorithm of Chand and Kapur [41] for computing general-dimensional convex hulls can construct Voronoi diagrams. The first gift-wrapping algorithm for constructing three-dimensional CDTs appears in a 1982 paper by Nguyen [159], but like the two-dimensional CDT paper of Frederick, Wong, and Edge [97], the author does not appear to have been aware of Delaunay triangulations at all. There is a variant of the gift-wrapping algorithm for CDTs that, by constructing the tetrahedra in a disciplined order and using other tricks to avoid visibility computations [200], runs in $O(nt)$ worst-case time, where n is the number of vertices and t is the number of tetrahedra produced.

Devillers [71] and Shewchuk [200] give gift-wrapping algorithms for deleting a degree-k vertex from a Delaunay or constrained Delaunay triangulation in $O(t \log k)$ time, where t is the number of tetrahedra created by the vertex deletion operation and can be as small as $\Theta(k)$ or as large as $\Theta(k^2)$. In practice, most vertices have a small degree, and naive gift-wrapping is usually fast enough.

The polygon insertion algorithm of Section 5.8 is due to Shewchuk [203]. Grislain and Shewchuk [105] show that it is NP-hard to determine whether a non-generic PLC has a CDT.

Exercises

1. You are using the incremental Delaunay triangulation algorithm to triangulate a cubical $\sqrt[3]{n} \times \sqrt[3]{n} \times \sqrt[3]{n}$ grid of n vertices. The vertices are not inserted in random order; instead, an

adversary chooses the order to make the algorithm as slow as possible. As an asymptotic function of n, what is the largest *total* number of changes that might be made to the mesh (i.e. tetrahedron creations and deletions, summed over all vertex insertions), and what insertion order produces that asymptotic worst case? Ignore the time spent doing point location.

2. Let S and T be two sets of points in $\mathbb{R}^3$, having s points and t points respectively. Suppose $S \cup T$ is generic. Suppose we are given $\mathrm{Del}\, S$, and our task is to incrementally insert the points in T and thus construct $\mathrm{Del}\,(S \cup T)$.

   We use the following algorithm. First, for each point p in T, find a tetrahedron or ghost tetrahedron in $\mathrm{Del}\, S$ that contains p by brute force (checking each point against each tetrahedron), and thereby build a conflict graph in $O(ts^2)$ time. Second, incrementally insert the points in T in random order, with each permutation being equally likely.

   If we were constructing $\mathrm{Del}\,(S \cup T)$ from scratch, with the insertion order wholly randomized, it would take at worst expected $O((s+t)^2) = O(s^2 + t^2)$ time. However, because S is *not* a random subset of $S \cup T$, the expected running time for this algorithm can be worse. An adversary could choose S so that inserting the points in T is slow.

   Prove that the expected time to incrementally insert the points in T is $O(t^2 + ts^2)$.

3. Extend the analysis of the randomized incremental insertion algorithm to point sets that are not generic. To accomplish that, we piggyback on the proof for generic point sets. Let S be a finite set of points in $\mathbb{R}^3$, not necessarily generic, and let $S[\omega]$ be the same points with their weights perturbed as described in Section 2.9. By Theorem 5.4, the randomized incremental insertion algorithm constructs $\mathrm{Del}\, S[\omega]$ in expected $O(n^2)$ time, and in expected $O(n \log n)$ time in the optimistic case where the expected complexity of the Delaunay triangulation of a random subset of the points is linear.

   When the randomized incremental insertion algorithm constructs $\mathrm{Del}\, S$, the tetrahedra and conflicts created and deleted are not necessarily the same as when it constructs $\mathrm{Del}\, S[\omega]$, but we can argue that the numbers must be comparable.

   (a) Show that immediately after a vertex v is inserted during the construction of $\mathrm{Del}\, S$, all the edges that adjoin v are strongly Delaunay. Show that immediately after v is inserted during the construction of $\mathrm{Del}\, S[\omega]$, the same edges adjoin v, and perhaps others do too.

   (b) The boundary of the retriangulated cavity is planar. Use this fact to show that the number of tetrahedra created when v is inserted during the construction of $\mathrm{Del}\, S$ cannot exceed the number of tetrahedra created when v is inserted during the construction of $\mathrm{Del}\, S[\omega]$.

   (c) Recall that each polyhedral cell of the Delaunay subdivision of S has its vertices on a common sphere, and that the cell is subdivided into tetrahedra in $\mathrm{Del}\, S[\omega]$. Show that the number of conflicts of a cell (vertices in its open circumball) in the Delaunay subdivision of S cannot exceed the number of conflicts of any of the corresponding tetrahedra in $\mathrm{Del}\, S[\omega]$. Show that the number of conflicts created when v is inserted during the construction of $\mathrm{Del}\, S$ cannot exceed the number of conflicts created when v is inserted during the construction of $\mathrm{Del}\, S[\omega]$.

4. Recall from Figure 3.14 that gift-wrapping can fail to construct the Delaunay triangulation of a point set that is not generic. One way to make the algorithm robust is to use symbolic weight perturbations. A different way is to identify groups of points that are cospherical during the gift-wrapping step and triangulate them all at once.

   Design a tetrahedral gift-wrapping algorithm that implements the second suggestion, without help from any kind of perturbations. Recall from Figure 4.4 that polyhedra of the Delaunay subdivision cannot be subdivided into tetrahedra independently of each other. How does your solution ensure that these subdivisions will be consistent with each other?

5. Give an example of a PLC $\mathcal{P}$ that has a CDT and a polygon f such that $\mathcal{P} \cup \{f\}$ is a valid PLC but does not have a CDT.