A faster circle-sweep Delaunay triangulation algorithm

Ahmad Biniaz and Gholamhossein Dastghaibyfard
Shiraz University, shiraz, Iran
biniaz@cse.shirazu.ac.ir, dstghaib@shirazu.ac.ir

August 13, 2011

Abstract

This paper presents a new way to compute the Delaunay triangulation of a planar set P of n points, using sweep-circle technique combined with the standard recursive edge-flipping. The algorithm sweeps the plane by an increasing circle whose center is a fixed point in the convex hull of P. Empirical results and comparisons show that it reduces the number of in-circle tests and edge-flips, and it is efficient in practice.

1 Introduction

Triangulation is a fundamental geometric structure in computational geometry. Given a point set P of n points in the plane, a triangulation $T(P)$ of P decomposes the convex hull of P into disjoint triangles whose vertices are exactly the sites of P. The most regular triangulation [32] of a set of points is the *Delaunay triangulation* (DT). It has the property that, whose triangles circumcircles contain no site in their interior. It maximizes the minimum angle of its triangles among all possible triangulations of a point set, and moreover maximizes lexicographically the *angle-vector* of whose triangles [4] (i.e. it limits the number of very narrow triangles). It also optimizes some other criteria—minimizing the largest circumcircle among the triangles, and minimizing a property called the *roughness* of the triangulation [32]. There are two extensions of Delaunay triangulation: *higher-order Delaunay triangulation* [20, 21, 5] and *weak Delaunay triangulation*[36]. DT has applications in terrain modeling and geographical information systems (GIS) [7, 8, 6, 19, 24], interpolation between points [32, 29], mesh generation for finite element methods (FEM) [12, 3], robotics, computer graphics [38], and etc.

This paper presents a new algorithm for the computation of DT. It uses the *sweep-circle* technique combined with the Lawson's [26] *recursive local optimization procedure*. The sweep-circle algorithm is based on the concept of wave front developed by Dehne and Klein [14], which uses an increasing circle emanating from a fixed point. The idea of the sweep-circle technique summarized as follows: firstly, the points are sorted according to their distances from a fixed point O in the convex hull of P. Then, it is imagined that a circle C centered at O increased and stops at some *event points*. A part of the problem being swept (inside the circle), is already solved, while the remaining part (out of the circle), is unsolved. The problem is completely solved when C passes through the last event point. This method is interesting notably when the triangulation has to be constructed locally around a given point [1]. This algorithm uses the *polar coordinates* of points, that is the coordinates derived from the distance and angular measurements from a fixed point (*pole*). Here, pole is the centre of C.

In addition, this paper presents an experimental comparison of proposed algorithm with the other popular plane sweep algorithms. Implementations of these algorithms are tested on a number of uniform and non-uniform data sets. We also analyze the major high-level primitives that algorithms use and do an experimental analysis of how often implementations of these algorithms perform each

operation. Experimental results show that the proposed algorithm is very fast, it is not sensitive to the distribution of input elements. It it is easy to understand and simple to implement. All of this, rates this algorithm among the best algorithms for constructing DT.

This paper is organized as follows: Section 2 gives a survey of existing algorithms for constructing DT and explores the plane sweep algorithms. In Section 3 the proposed algorithm is explained in detail. Section 4 gives experimental results and comparisons, and we conclude this paper in Section 5.

2 Background

There are many sequential algorithms for the construction of DT. We classify them according to Su and Drysdale [37] into five categories:

- *Divide and conquer (D&C) algorithms* - these algorithms are based on recursive partitioning and local triangulation of the point set, and then on a merging phase where the resulting triangulations are joined. The recursion usually stops when the size of the point set matches some given threshold. Local triangulation is then constructed by an algorithm belonging to any of the next categories [27, 16, 23].

- *Incremental insertion algorithms* - these algorithms insert the points in P one by one: the triangle containing the point to be inserted is subdivided and then the circumcircle criterion is tested recursively on all triangles adjacent to the new ones and if necessary, their edges are flipped. It is simpler to starting with an auxiliary triangle that contains all points in its interior [4, 22, 25, 40].

- *Gift-wrapping algorithms* - starting with a single Delaunay triangle and then incrementally discovering valid *Delaunay triangles* [34], one at a time. Each new triangle is grown from an edge of a previously discovered triangle by finding the site that joins with the endpoints of that edge to form a new triangle whose circumcircle is empty of sites [15, 28].

- *Convex hull based algorithms* - these algorithms transform the points into E^3 (three-dimensional space) and then compute the convex hull of the transformed points. The Delaunay triangulation is obtained by projecting the resulting convex hull back into E^2 (two-dimensional space) [10, 2].

- *Plane sweep algorithms* - the sweep line (or plane sweep) is one of the most popular acceleration techniques used to solve 2D geometric problems [31]. Firstly, the points are sorted. Then, it is imagined that the sweep-line glides over the plane and stops at event points. The sweep-line can move in the y, x, or any other directions in the plane. The part of the problem being swept is already solved, while the remaining part is unsolved. The problem is completely solved when the sweep-line passes through the last event point [39, 18].

The main objective of this paper is about plane sweep algorithms for computing DT. Reference [9] presents a survey on plane sweep DT algorithms. Here, we give an overview about three popular algorithms in this category:

- Fortune's [18] sweep-line algorithm which adds a Delaunay triangle to the triangulation at some event points,

- Žalik's [39] sweep-line algorithm which is based on *legalization* [4], and

- the sweep-circle algorithm proposed by Adam et. al. [1], which adds *validated* Delaunay edges and regions to the diagram at some event points.

2.1 Fortune's sweep-line algorithm

In 1987, Fortune [18] finds an $O(n \log n)$ scheme for applying the sweep-line approach to construct DT in the plane. The algorithm maintains two set of states. The first is a sweep-line state that is an ordered list of sites called the *frontier* of the diagram; Figure 1 shows that the entry of site s corresponds to an interval I_s on the sweep-line where each maximal empty circle with topmost point in I_s touches site s [17]. The second is a priority queue, called the *event queue*, used to determine the next sweep-line move (place where the sweep line should stop). This queue stores two type of event points called *site events* and *circle events*. Site events happen when the sweep-line reaches a site and circle events happen when it reaches the top of the circle formed by three consecutive vertices on the frontier (see Figure 1).

The algorithm updates the sweep-line data structure and event queue when the sweep-line passes through a circle event and discovers a Delaunay triangle.

In experiments, we used the version of this algorithm that implemented by Shewchuk in *Triangle* package [35]. In this version the frontier is implemented as a splay tree that stores the random sample of roughly one tenth of the boundary edges. When the sweep-line passes through an input point, this point must be located relative to the boundary edges; this point location involves searching in splay tree, followed by a search on the boundary of triangulation itself. To represent the priority queue, an array-based heap is used. Members of queue are inserted to heap according to their priorities, so finding the next event point involves extracting the minimum event point from the heap.

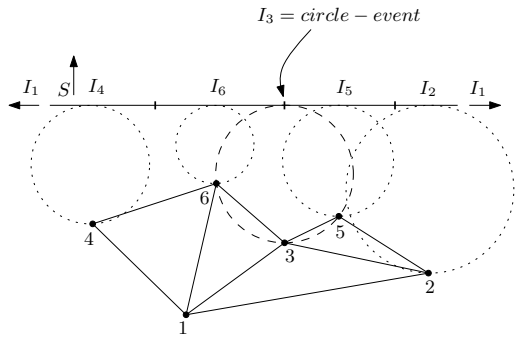

Figure 1. Points are numbered according to their priorities. The frontier is (1, 4, 6, 3, 5, 2, 1). The next Delaunay triangle is $\triangle_{6,3,5}$.

2.2 Žalik's sweep-line algorithm

Žalik [39] finds a fast $O(n \log n)$ expected time algorithm for constructing 2D DT. It is based on the combination of sweep-line paradigm and legalization criterion. The algorithm surrounds the swept vertices by two bordering polylines. The first is the upper polyline that is represented by a so-called *advancing front*, the second is the lower border forms part of the convex hull, which is called *lower convex hull*. All vertices between the lower convex hull and the advancing front are triangulated according to the empty circle property.

The main idea of the algorithm summarized as follow: the first triangle(s) is constructed and its vertices are oriented in a counter-clockwise direction, then advancing front and the lower convex hull are initialized. According to the collinearity or non collinearity of the first $m(m \geq 3)$ points, 15 possible configurations may appear. Then, the sweep-line moves and when it meets the next site, a vertical projection of that site is done on the advancing front. According to the hit or miss of the advancing front, four possible cases (HIT, ON-EDGE, LEFT and RIGHT) may appear. Figure 2.a

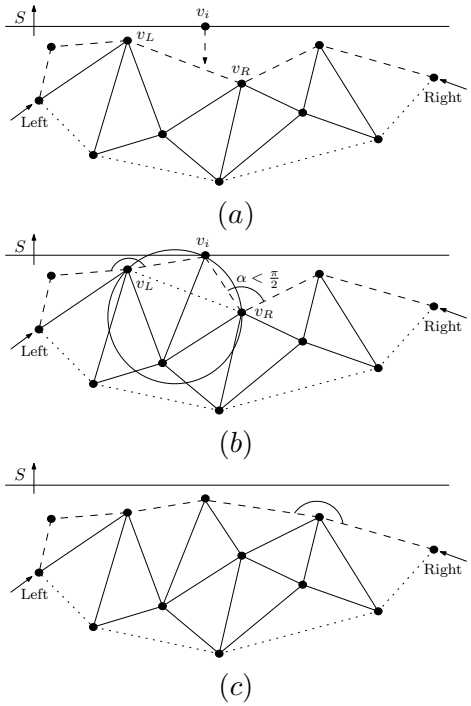

Figure 2. The advancing front edges in dashed lines and the lover convex hull edges in dot lines: vertical projection of the next site hits the advancing front (a), new triangle is constructed and legalized, advancing front is updated, walking to the left and right (b), and walk in right-side direction is stopped (c).

shows the most common case, when the projection hits the advancing front. With three sites v_i, v_L and v_R, a new triangle is constructed and checked recursively with the neighboring triangle according to the empty circle property (Figure 2.b). Then, the algorithm walks to the left and right on the advancing front and new triangles are constructed and legalized while the outside angle between three consecutive points is smaller than $\pi/2$ (Figures 2.b and 2.c). In this way, the *basin* [39] may appears, which is triangulated by monotone polygon triangulation [4].

2.3 Adam's et. al. sweep-circle algorithm

Reference [1] introduced an algorithm that computes the Delaunay diagram [17] by sweeping the plane with an increasing circle C, centered at O. The algorithm maintains the sweep-circle data structure by an ordered list of sites called the *frontal* (*front*) of the diagram. The algorithm also updates the current Delaunay diagram by processing two kind of event points: *site event* and *ultimate point event*. Site events happen when the sweep-circle reaches a site, and ultimate point events happen when it reaches the farthest point of a circle which is formed by three consecutive frontal sites (Figure 3). Site events [resp. ultimate point events] add validated Delaunay edges [resp. regions] to the diagram. An edge $e(s, t)$ [resp. region] is said to be validated if there exists a circle contained in C that goes through s and t [resp. all sites of region] and contains no site of P in its interior.

When C increases and sweeps over a site s, the algorithm searches a site t in the front such that s and t can be linked together, to form a validated Delaunay edge. To locate the point s in the frontal edges, a balanced binary search tree is used. A frontal edge is inserted in tree when created and is removed when it is no longer frontal. To be able to find the ultimate point closest to O, the ultimate points are inserted into another balanced binary search tree according to their distance from O. An ultimate point is inserted in the tree when created and is deleted from the tree when killed.

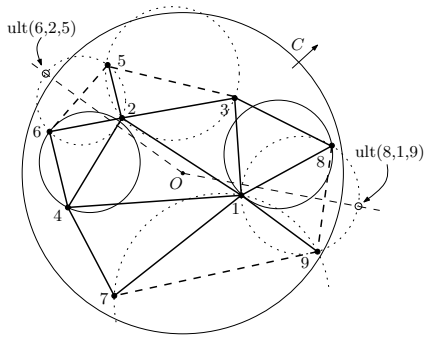

Figure 3. The validated Delaunay edges in full-lines and the non-validated edges in dashed lines. The sites are numbered according to their distance from O. Front is (1, 9, 1, 7, 4, 6, 2, 5, 3, 8, 1). The ultimate points associated to frontal triples (6, 2, 5) and (8, 1, 9) are shown.

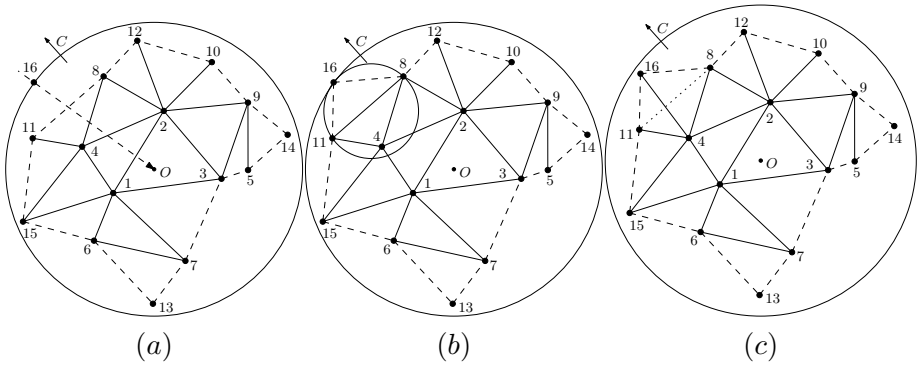

Figure 4. The main idea of the sweep-circle algorithm. The sites are numbered according to their distances from O. Projection of v_{16} hits the edge (v_{11},v_8) of the frontier (a), new triangle is generated (b) and legalized (c).

There are n site events. For every ultimate point event, a region is updated, and since Delaunay diagram of P is a partition of the plane with n vertices; it admits at most $2n-4$ regions. Thus the algorithm handles at most $3n-4$ event points. Every search, insertion or deletion in balanced binary search tree is done in $O(\log n)$ time. Therefore the construction of the Delaunay diagram of P is done in $O(n \log n)$ time.

3 Proposed algorithm

The main idea of the algorithm is summarised in Figure 4. Points are processed according to their increasing distance from the pole O, which is the sweep-circle centre. Figure 4.a shows the configuration when the sweep-circle C, has already passed through the first 15 points. The algorithm surrounds them by a bordering polygon (plotted by dashed lines), which is called the *frontier* of the diagram. The frontier separates the swept vertices from the non-swept. The shape of the frontier depends on the arrangement of the points already passed and, in general, does not coincide with the convex hull of swept vertices. All vertices inside the frontier are triangulated with respect to the empty circle property. In this algorithm, the newly encountered point, v_{16}, is projected onto the frontier toward O. It hits the edge (v_{11}, v_8). With the point v_{16} and the edge (v_{11}, v_8) a new triangle $\triangle_{16,8,11}$ is created and the frontier is changed to correspond to the new situation (see Figure 4.b). This new triangle tested with its neighboring triangle $\triangle_{4,11,8}$ according to the empty circle property. As seen from Figure 4.b, it fails the test, and "legalized" using standard recursive edge-flipping. As a result,

triangulation of first 16 points according to the empty circle property is obtained inside C (see Figure 4.c). After all points have been swept, the frontier is not convex and does not correspond to the convex hull. Additional triangles have to be added.

The projection of new points toward pole which used in the sweep-circle algorithm, eliminates the special cases where the vertical projection in Žalik's sweep-line algorithm can miss the frontier. It also decreases the expected number of edge-flipping. We also provide a slightly improved method for detecting *basins*. Now, we describe the algorithm in more details. The algorithm computes DT in three phases: *initialization*, *triangulation* and *finalization*.

3.1 Initialization

Assume that the 2D set $P = v_1(x_1, y_1), v_2(x_2, y_2), ..., v_n(x_n, y_n)$ shows the input dataset. The initialization of the algorithm includes:

1. selecting the origin of the polar coordinates O,
2. calculating the polar coordinates of input points,
3. sorting input points in increasing distances from O,
4. construction of the first triangle, and
5. determination of the initial frontier.

Firstly, the origin of the polar coordinates $O(p_x, p_y)$ (centre of the sweep-circle C) is selected such that, be inside the first triangle. We select p_x (resp. p_y) as the average of the largest and smallest x (resp. y) of input points. Then, we calculate the polar coordinates of the input points. Each point $v_i(x_i, y_i)$ in Cartesian coordinates can be transformed to $v_i(r_i, \theta_i)$ in polar coordinates by:

$$r_i = \sqrt{(x_i - p_x)^2 + (y_i - p_y)^2} \tag{1}$$

$$\theta = \begin{cases} \arccos(\frac{x_i - p_x}{r_i}) & if(y_i - p_y) > 0 \\ \pi + \arccos(\frac{x_i - p_x}{r_i}) & if(y_i - p_y) < 0 \end{cases} \tag{2}$$

The location of the origin may has a little influence on the number of flips and consequently on the runtime of the algorithm. The tests show if the points are uniformly distributed in gyrate regions with centre O, the algorithm runs a little faster (see the results in subsection 4.2 for Gaussian dataset).

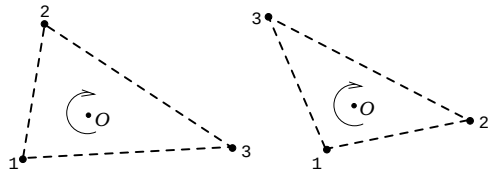

Figure 5. Two possible configurations for initial frontier. The sites are numbered according to their distances from O.

Now, the points are sorted according to their r-coordinate. If two points have the same r-coordinate, they are sorted according θ as the second criterion. Then, the first three points are taken for the first triangle. These three points are oriented in a clockwise direction and the first triangle is constructed. Then, the frontier is initialized; two possible configurations, $(1, 2, 3)$ or $(1, 3, 2)$,

may appear (see Figure 5) instead of 15 possible configurations in Žalik's sweep-line algorithm [39]. In the special case when the first point coincides with the origin O (its r-coordinate is zero), we remove that point from the list and compute the triangulation. Finally, after finishing the triangulation, we insert that point to the triangulation; the process is very simple [4].

3.2 Triangulation

When C increases and sweeps over a site v, we project it on the frontier toward O. This projection always hits the frontier because O lies inside the frontier and v lies outside. Thus we only have the HIT case. Figure 6.a shows the situation when the projection of the next vertex v_{18} hits the edge (v_3, v_7) of the frontier. A new triangle $\triangle_{18,7,3}$ is constructed and checked with its neighbor for Delaunay criteria. The vertex v_{18} located in the frontier according to its θ-coordinate and inserted between vertices v_3 and v_7 on the frontier (Figure 6.b). If the projection of the new vertex hits a vertex s on the frontier, with this two vertices and the vertex at the left (or right) of s on the frontier a new triangle is constructed. Usually, the new vertex also has to be connected with other vertices of the frontier. To do this, two heuristics have been introduced by Žalik [39] to prevent the construction of tiny triangles, which would be legalized. The first is walking to the left-side and to the right-side on the frontier, and the other is removing basins.

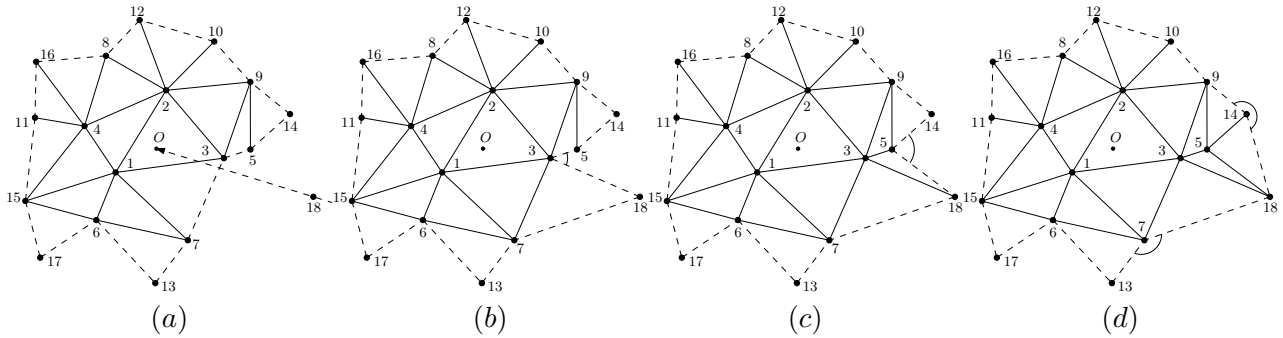

Figure 6. The next unused vertex v_{18} is taken from the input list. Projection of v_{18} hits the edge (v_3, v_7) of frontier (a), walking to the left side started (b), walking on the left follower site (c), end of left-side walking (d).

The first heuristic for left-side walk is done as follows: If the angle between the vectors determined by vertices v_{18}, v_3 and v_5 (Figure 6.b) is smaller than $\pi/2$, the new triangle $\triangle_{18,3,5}$ is generated (Figure 6.c), otherwise the walk in the left-side direction is stopped. The new triangle is checked recursively against the Delaunay criteria with the two neighboring triangles ($\triangle_{18,7,3}$ and $\triangle_{5,3,9}$). The vertex v_3 is removed from the frontier (Figure 6.c) and the process is repeated for vertices v_{18}, v_5 and the left follower vertex on the frontier (v_{14} in our example) until the angle between them become greater than $\pi/2$ (Figure 6.d). Thus, walking to the left side is stopped and walking to the right side is started symmetrically (Figure 6.d). Unfortunately, in this way the frontier may becomes considerably wavy leading, in some cases, to what is called basins (Figure 7.b). If no point appears upon some parts of the frontier for a while, then a curve may be formed on the frontier which is called basin [39]. When one point finally appears, tiny triangles are generated and then legalized by several diagonal swaps (Figure 7.a). If a dataset has some basins, many tiny triangles are generated that must be legalaized; legalization is one of the most time consuming operation during the algorithm (see the experiments). Thus, it slows down the algorithm for that dataset and the behaviour of the algorithm is not normal for such datasets, so the running time of the algorithm is sensitive to the distribution of the input points. We introduce two heuristics to detect basins early enough and remove them. The first procedure is as

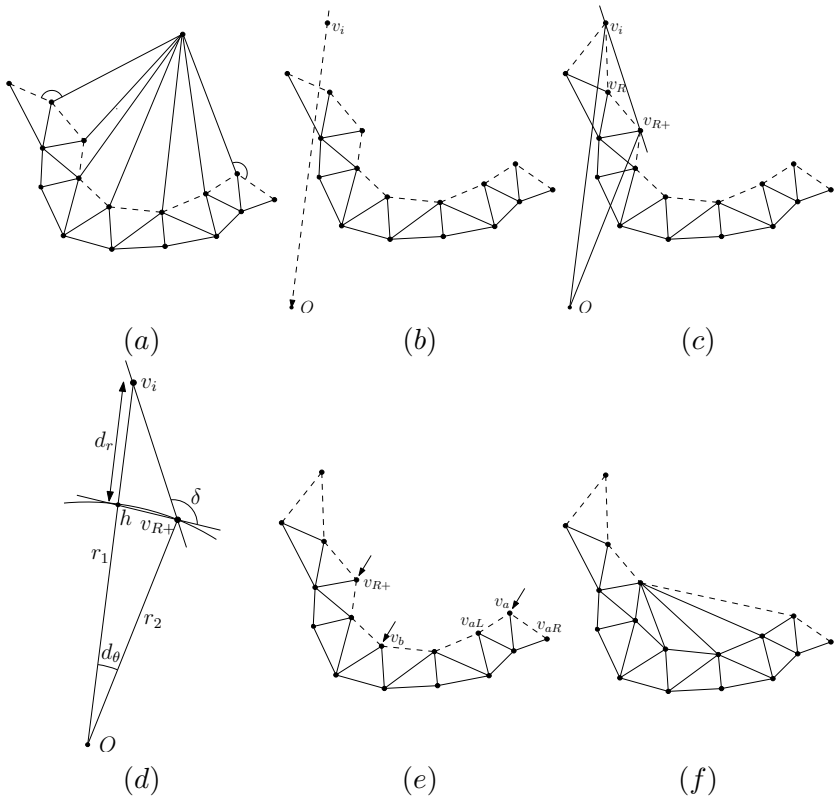

Figure 7. The basin. Generation of tiny triangles(a), Projection of v_i on the frontier (b), right-side and right-side follower neighbors of v_i (c), detecting the basin (d), bottom vertex, left and right borders of basin (e), and removing basin(f).

follows: we consider the procedure while detecting the right-side basin; the solution for the left side is symmetrical. Let vertex v_R be the right-side neighbor of vertex v_i in the frontier, and let v_{R+} be its right-side follower (Figure 7.c). Let the r-coordinate of v_i be r_1 and the r-coordinate of v_{R+} be r_2, and let the arc with centre O and radius r_2 hits r_1 in h (it is easy too show that: $r_2 \leq r_1$), (see Figure 7.d). If the angle δ is smaller than $3\pi/4$, a basin is found (as the condition in [39]). The second procedure which is experimentally better than the former is as follows: let d_r and d_θ be the difference between r and θ coordinates of vertices v_i and v_{R+} (see Figure 7.d). This Figure shows that the occurrences of basin has direct relation to d_r and indirect relation to d_θ. So the value of $d_r/(r_2 d_\theta)$ is determined as to whether it is greater than 2. In this case a basin is found. In this formula, division by r_2 is for normalizing the effect of d_r. The value of 2 is selected experimentally. Basins founded in this way decrease the number of tiny triangles even more than previous heuristic; reducing the spent CPU time (see the analysis in subsection 4.4). Finally we remove the basin by monotone polygon triangulation [4] as follows: The vertex v_{R+} is considered as the left-side border of the basin. The bottom vertex of the basin, v_b, which has locally minimum r-coordinate, is found first starting search from v_{R+} until r-coordinate starts to increase and then the right-side border vertex, v_a, is searched for on the other side, which the signed area of v_{aL}, v_a and v_{aR} is negative (see Figure 7.e). Now, we have a list of vertices between v_{R+} and v_a, which sorted according to their r-coordinate. Then select the first three vertices of the list; create a triangle with them and remove a vertex from the list which is not on the basin any more, and iterate this procedure. Each generated triangle is checked recursively against the empty circle property with their neighbors. Finally, the frontier connects the borders of the basin, and in this way it is smoothed considerably (Figure 7.f).

3.3 Finalization

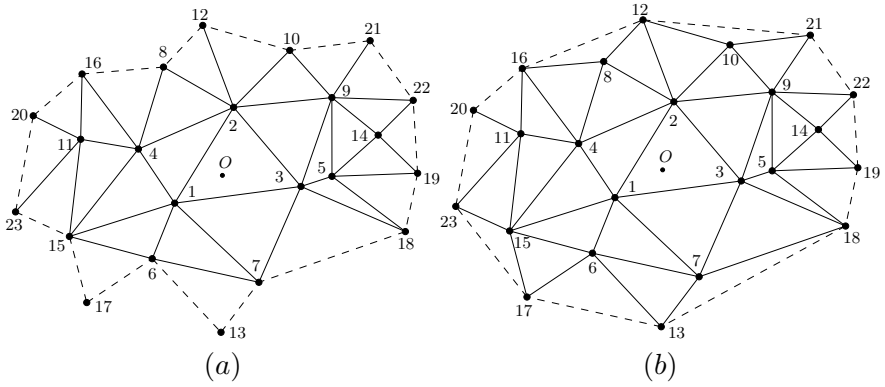

Figure 8. Conversion of the frontier to the convex hull. Scanning the frontier for missed triangles (a), frontier is now coinciding with the convex hull, and triangulation is completed (b).

After sweeping all points, the front may not be convex, because it is usually jagged, with triangles jutting out and gaps between them (see Figure 8.a). The missed triangles are added by performing a clockwise walk through the edges of the frontier. Three consecutive vertices of the frontier determine a missed triangle if they have a left turn, for example vertices v_{16}, v_8, and v_{12}. The middle vertex (vertex v_8 in Figure 8.a) is deleted from the frontier. The next triple (v_{16}, v_{12}, v_{10}) shows a right turn and the algorithm moves a step further and considers the vertices (v_{12}, v_{10}, v_{21}) and then (v_{12}, v_{21}, v_{22}) . The complete Delaunay triangulation is shown in Figure 8.b.

Algorithm 1 summarizes the main idea of the proposed sweep circle algorithm that computes the Delaunay triangulation of the input point set P. Experiments in Section 4 show that this algorithm is more efficient from other popular DT algorithms.

Algorithm 1. Sweep circle Delauany triangulation algorithm

SWEEP-CIRCLE-DT(P):

Input: A 2D set $P = v_1, v_2, ..., v_n$ of n points

Output: DT(P)

Initialization:

1: select the pole O
2: calculate (r, θ) for points in P
3: sort the P according to r
4: create the first triangle
5: initialize the frontier

Triangulation:

6: **for** $i \leftarrow 4$ **to** n **do**
7: project v_i on the frontier; hits the edge (v_L, v_R)
8: create triangle $\triangle_{i,L,R}$ and legalize it recursively
9: insert v_i between v_L and v_R in the frontier
10: walk to the left on the frontier until stop at v_{L+}
11: walk to the right on the frontier until stop at v_{R+}

12: solve the basin that may formed by v_i and v_{L+}

13: solve the basin that may formed by v_i and v_{R+}

Finalization:

14: scan the frontier for the missed triangles

---

4 Analysis of the algorithm

The frontier data structure must support $O(1)$ time access to a triangle from a neighboring triangle. Almost all commonly used data structures storing planar triangulations support this operation: DCEL (Doubly Connected Edge List) or simply a linked list of triangles with all the necessary local information (three vertices, three edges, and three pointers pointing to its neighboring triangles). The frontier is formed of boundary points which always sorted according to their θ-coordinate. Each vertex of the frontier points to the rightmost triangle being defined by that vertex (see Figure 9). In this way, triangles touching the frontier with one or two edges are accessed directly.

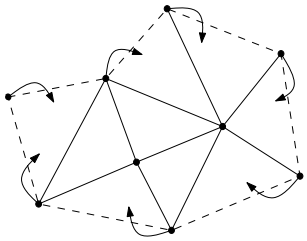

Figure 9. Each vertex of the frontier points to the rightmost triangle being defined by that vertex.

The frontier can be implemented by any range searching data structure such as a heap, or balanced binary search trees (e.g. AVL tree, B-tree, Read-Black tree [13]). In our case, a simple hash-table on a circular double linked list is used (Figure 10). Each record of the frontier corresponds to a vertex and stores the index of that vertex Vi, and the index of the rightmost triangle Ti defined by that vertex and sharing its edge with the frontier. Records are sorted regarding the θ-coordinates of vertices.

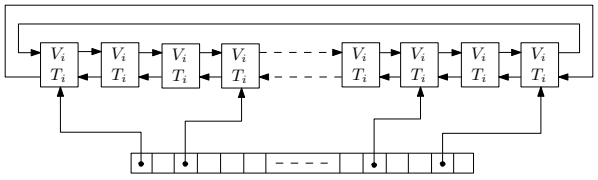

Figure 10. Implementation of the frontier: A hash-table on a circular double linked list.

We use the formula in [39] to determine the number of entries into the hash-table:

$$h = 1 + \lfloor n/k \rfloor \tag{3}$$

Where h is the size of hash-table, n is the number of input points and k is a constant factor that is determined before the triangulation starts. By the following fact [30] we can show that the size of frontier in proposed algorithm is bigger than the size of advancing front in sweep-line algorithms:

Fact 1 *The average number of vertices of the convex hull of a set of n sites that randomly distributed in a circle is in* $O(\sqrt[3]{n})$ *but this number is in* $O(\log n)$ *for sites that randomly distributed in a rectangle.*

We analyzed the relation between spent CPU time and the number of entries into the hash-table, h, which is related to k. Figure 11 compares the spent CPU time of our sweep-circle and Žalik's sweep-line algorithms according to parameter k, while triangulating different set of points at uniform distribution. The same results also obtained by triangulating non-uniform distributions such as Gaussian and grid datasets. It shows that the value of k has a great affect on the running time of both algorithms. If k is too small or too large, the running time of the algorithms increased. So, it is important to select an appropriate value for k.

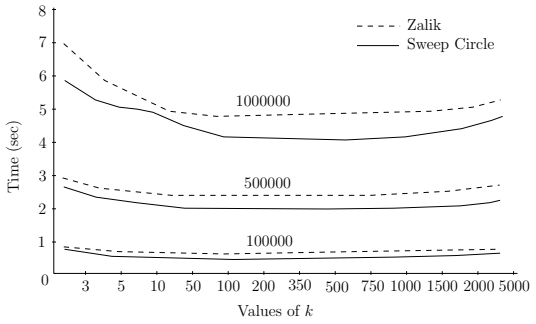

Figure 11. The influence of parameter k on spent CPU time in triangulation phase. Dashed lines for Žalik, and solid lines for sweep-circle.

When sweep-circle sweeps an input point, the hash-table entry for that point calculated firstly, in time $O(1)$. Then, the point must be located to the frontier to identify the hated edge by the projection; this point location involves searching in hash table for the first non-NULL entry followed by a search in a portion of circular double linked list that assigned to that entry. We called these two types of searches, *table-search* and *list-search*, respectively. The algorithm was tested on uniform inputs with size 100 000, 500 000, and 1 000 000 for different values of k. Figure 12 shows that there is a trade off between the number of table-search and the number of list-search. It also shows that any value between $\sqrt[3]{n}$ and $\sqrt[2]{n}$ for parameter k gives very acceptable results (see also Figure 11). Žalik [39] used $k = 100$ and we use $k = \sqrt[3]{n}$ optionally. In this case the total number of searches (table-searches plus list-searches) per site is at most 10 regardless of the number of input vertices. Worse results are obtained only if k is close to 1 or too large (see Figures 11 and 12). If k is close to 1, then there are too many entries into the hash-table, many entries are empty during sweeping. The wasting of time for table-search significantly influences the total CPU time spent. On the other hand if k is too large, then there are smaller number of entries into the hash-table, many points are assigned for each entry of the hash-table. This increases the spent CPU time for list-search.

Remain of this section analyze the algorithm from the following aspects:

- time and space complexity analysis,
- spent CPU time comparisons using artificial data sets, and
- CPU time-independent comparisons.

4.1 Time and space complexity

The complexity of initial phase of the algorithm is estimated as follows: the polar coordinates of all the n input points are calculated in $O(n)$ and then, they are sorted according to their r-coordinates

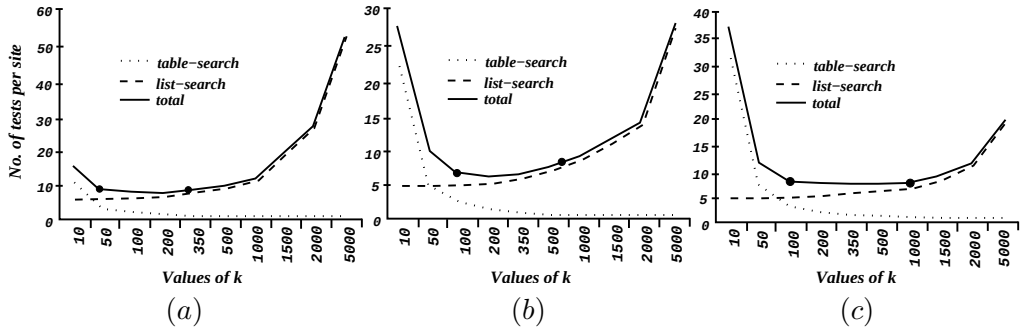

Figure 12. Number of searches per site while triangulating 100 000 (a), 500 000 (b), and 1 000 000 (c) uniformly distributed points for different values of k. Total searches (table-searches + list-searches) are marked in $k = \sqrt[3]{n}, \sqrt[2]{n}$.

by Quicksort in $O(n \log n)$, thus the total time complexity of initialization phase is:

$$T_{initial}(n) = O(n) + O(n \log n) = O(n \log n). \tag{4}$$

In the triangulation phase, the new point located to the frontier and new triangles are generated and legalized. After the insertion of ith vertex, we denote the size of frontier by $3 \leq f_i \leq n$. So, there are f_i elements that assigned to the hash table of size h. Each element is assigned to h/f_i entries of the hash table and to find the first non-NULL entry h/f_i entries must be searched. On the other hand, f_i/h elements are assigned to each entry of the hash table; consequently the size of linked list related to that entry is f_i/h. So, to locate the new point to the frontier, h/f_i table-search and f_i/h list-search are done on average. Thus, the total number of searches for all sites is expected to be:

$$T_{locate}(n) = \sum_{i=4}^{n} (\frac{h}{f_i} + \frac{f_i}{h}) \tag{5}$$

Since $k = \sqrt[3]{n}$, thus $h = 1 + \lfloor n/k \rfloor = O(n/\sqrt[3]{n}) = O(\sqrt[3]{n^2})$. But for different distributions a different number of points lie on the frontier and at each iteration of the algorithm the precise value of f_i depends on the arrangement of the points already passed. The frontier has a wavy shape and does not coincide with the convex hull, thus, it has more vertices than convex hull of swept points. So, the size of frontier is bigger than $O(\sqrt[3]{i})$ using Fact 1. To estimate the average size of frontier we perform an experiment. The algorithm was tested on four different distributions of points with sizes ranging from 2^{10} to 2^{20} sites. Figure 13 shows the size of frontier for each distribution. To estimate each curve by a formula, we used regression analysis and estimated each curve with some different terms. The greatest term was $\sqrt[3]{i^2}$ for all datasets, but the other terms (which has no affect in complexity analysis and their sign is not important) was stated by big-O notation. By removing the less important terms the size of frontier for each dataset is $f_i = O(\sqrt[3]{i^2})$ on average. Thus

$$\begin{aligned} T_{locate}(n) &= \sum_{i=4}^{n} (\frac{h}{f_i} + \frac{f_i}{h}) \\ &\leq h \sum_{i=1}^{n} \frac{1}{f_i} + \frac{1}{h} \sum_{i=1}^{n} f_i \\ &= \sqrt[3]{n^2} \sum_{i=1}^{n} \frac{1}{\sqrt[3]{i^2}} + \frac{1}{\sqrt[3]{n^2}} \sum_{i=1}^{n} \sqrt[3]{i^2} \end{aligned}$$

$$
\begin{aligned}
&\leq \sqrt[3]{n^2} \int_0^n x^{-\frac{2}{3}} dx + \frac{1}{\sqrt[3]{n^2}} \int_1^{n+1} x^{\frac{2}{3}} dx \\
&= 3n + O(\frac{3}{5}n) = O(3.6n) \\
&= O(n). \qquad (6)
\end{aligned}
$$

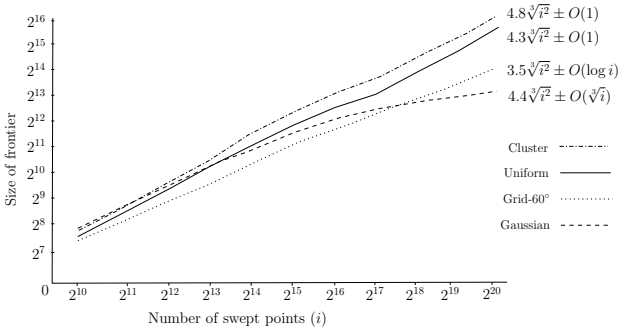

Figure 13. Size of frontier.

Where, the summation is over all the number of swept points. This equation shows that the total number of searches per point is 3.6 on average. It is interesting that the result of this equation is similar to the results of Figure 12 and Table 6; the total number of searches per site in Figure 12 and Table 6 is a constant value between 3 to 14, regardless of the number of input vertices.

Generation of the new triangles and checking for the introduced heuristics are done in constant time. The legalization is terminated in logarithmic time per site [4]. Thus, the total time of the algorithm's second part is therefore:

$$
\begin{aligned}
T_{triang}(n) &= T_{locate}(n) + O(n \log n) \\
&= O(n) + O(n \log n) \\
&= O(n \log n) \qquad (7)
\end{aligned}
$$

The time complexity of the final part of the algorithm that is completion of the triangulation is relevant to the size of final frontier:

$$
T_{final}(n) = O(\sqrt[3]{n^2}) \qquad (8)
$$

Thus, the common expected time complexity of the proposed algorithm is:

$$
\begin{aligned}
T_{common}(n) &= T_{initial}(n) + T_{triang}(n) + T_{final}(n) \\
&= O(n \log n) + O(n \log n) + O(\sqrt[3]{n^2}) \\
&= O(n \log n) \qquad (9)
\end{aligned}
$$

Now, let us estimate the space complexity of the algorithm: n memory locations are assigned to the input points and at most $2n - 5$ memory locations are reserved for triangles [4]. The hash-table contains at most n entries. Roughly $O(\sqrt[3]{n})$ records are needed for the frontier. Therefore, the total space complexity of the algorithm is $O(n)$.

4.2 Comparing spent CPU time

This section gives an analysis of the proposed algorithm's spent CPU time and compares it with the others. The run time of proposed algorithm (besides calculating the polar coordinates of the input points and sorting them) is proportional to the cost of searching, updating the frontier, and updating Delaunay diagram when a new triangle is created.

We implemented the proposed sweep-circle (SC) algorithm, and compare it with the following popular Delaunay triangulation algorithms:

- Incremental Delaunay triangulation library of CGAL-3.1 [11]. It allows the insertion of new points later on without recomputing everything from scratch.
- The Steven Fortune's sweep-line algorithm (FS) [18], which implemented by Shewchuk within a triangulation package, Triangle [35, 33]. Adam et. al. [1] compared their sweep-circle algorithm with Fortune's algorithm and showed that the implementation of their algorithm is almost 20% slower than the sweep-line algorithm.
- The improved version [16] of Guibas and Stolfi's [23] divide-and-conquer algorithm. Guibas and Stolfi [23] gave an $O(n \log n)$ Delaunay triangulation algorithm, which uses the in-circle test. Dwyer [16] showed that a simple modification of this algorithm runs in $O(n \log \log n)$ expected time on uniformly distributed sites, by alternating between vertical and horizontal cuts. In 1995, Su and Drysdale [37] showed that Dwyer's algorithm is strongest among other popular algorithms. Rex Dwyer provided code for his divide-and-conquer algorithm (GSD).
- Žalik's sweep-line algorithm (ZS) [39]. Borut Žalik provided the code for his sweep-line algorithm and compared it with other Delaunay triangulation algorithms such as randomized incremental [22, 25, 40], divide-and-conquer [27] and Fortune's sweep-line [18] and showed that his algorithm is the fastest.

Some notes on implementations are in order. First, all algorithms implemented with C/C++ and test the in-circle primitive by computing the sign of a determinant. All algorithms use floating point computations, except CGAL and Shewchuk's implementation of sweep-line which use exact arithmetic operations. Using exact arithmetic comes at some cost and could changes the relative run times. We tried to share operations and data structures when this was possible.

We test the algorithms on a PC with Intel Pentium 4, 1.6 GHz processor and 512 MB of RAM running under the Microsoft Windows XP Professional Version 2002 operating system. We install CGAL on Windows using Cygwin. Input point sets are used with uniform and Gaussian distributions and points arranged in clusters and tilted grids (Figure 14). For each data size we tested several different data sets with the same distribution. Time for I/O operations (i.e., reading points from input file into the memory and storing the resulting triangulation onto the disk) is excluded.

Tables 1 and 2 compare the total run times of algorithms (Table 2 compares only the plane sweep algorithms). The proposed algorithm, SC, is the fastest for all distributions, Žalik's and Dwyer's algorithms stand second and third respectively. Dwyer's algorithm spends most of its time for in-circle test and Fortune's algorithm spends most of its time in searching event queue. We remark that Shewchuk puts a lot of effort into making his code numerically stable. The CGAL implementation performs poorly, wasting of time for locating new point in the triangulation. For our algorithm, points arranged in regular grid represent the best case and points with uniform distribution represent the worst case. For points arranged in tilted grid, the reaction of algorithms is very strange and shown in Table 2. When grid is tilted near 0, 45 and 90 degrees, there are numerous points with the same y-coordinate. In sweep line algorithms, at each time that sweep line moves, it reaches all of those

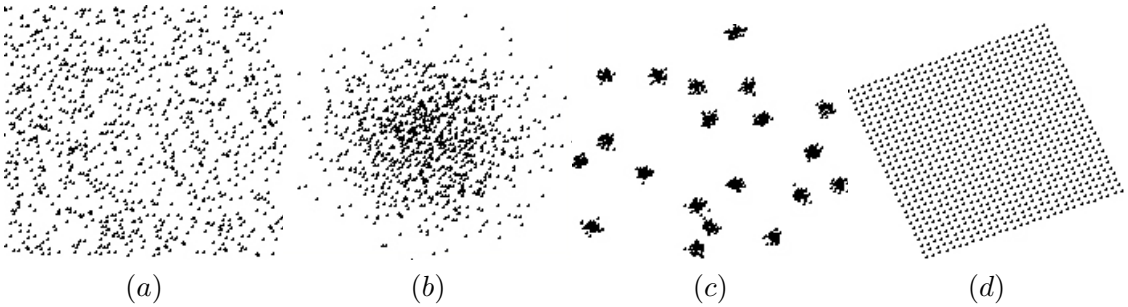

Figure 14. Various point distributions: uniform (a), Gaussian (b), points arranged in clusters (c), and points arranged in tilted grid (d).

points simultaneously. In this case the sweep line has either a horizontal straight line shape or regular wavy shape, which decreases the number of flips. But in other degrees, the sweep line has an irregular wavy shape, which increases the number of flips and runtime of the algorithm. This claim is also true for sweep circle. It is interesting that sweep circle has no diagonal swapping for grids that tilted near 0, 45, and 90 degrees.

Table 1. Timings (in second) for triangulation, not including I/O. Input points chosen from one of three distributions: uniformly distributed random points in a square, Gaussian points, and points arranged in clusters.

No. of points	100 000			500 000			1 000 000		
Distribution	Unf	Gus	Clu	Unf	Gus	Clu	Unf	Gus	Clu
CGAL	4.34	4.22	3.02	54	41	13	162	122	63
FS	2.23	2.20	1.76	15.1	14.5	11.3	33.2	30.6	25.2
GSD	1.18	1.22	1.36	3.37	3.73	4.11	7.21	8.35	9.78
ZS	0.43	0.44	0.44	2.22	2.27	2.21	4.59	4.63	4.52
SC	0.35	0.33	0.37	1.98	1.84	1.88	4.32	4.12	3.81

Table 2. Strange run times (in second) of algorithms when triangulating 1 000 000 points arranged in k degree tilted 1000×1000 square grids.

k	0	10	20	30	40	45	50	60	70	80	89.5
FS	21.1	29.9	29.9	28.8	29.3	19.7	28.2	28.1	30.9	27.1	19.6
ZS	3.44	failed	3.49	3.47	3.51	3.62	3.46	3.60	3.43	3.51	6.19
SC	3.01	3.27	3.23	3.26	3.24	3.03	3.32	3.50	3.30	3.45	3.05

Table 3 compares the spent CPU time on initialization and on the triangulation phase within Žalik's sweep-line and proposed sweep-circle algorithms. Initializations in Žalik involves sorting input points by Quicksort, but in sweep-circle it involves calculating the polar coordinates of input points, followed by sorting them using Quicksort. In Žalik 23% of total time is required for sorting, but in sweep-circle 32% of total time is required for polar coordinates calculation and sorting (see also Figure 15). Figure 15 shows typical run times needed for all parts of sweep-circle and Žalik algorithms. In sweep-circle, 7-9% of time is spent for calculating polar coordinates of the input points. Thus, if the polar coordinates of input points are known in advance, the running time of the algorithm decreased even more than the run times that shown in Tables 1, 2 and 3.

At the end, the performances of the all referenced algorithms are tested and compared according to some real world data sets with different size. The points are obtained from the boundary of objects

Table 3. Spent CPU time (s) within Žalik and sweep-circle while triangulating point sets with uniform distributions.

No. of points	100 000		500 000		1 000 000	
Algorithm	Žalik	SC	Žalik	SC	Žalik	SC
Initialization	0.094	0.114	0.52	0.62	1.16	1.327
Triangulation	0.337	0.236	1.70	1.36	3.46	3.00
Total	0.431	0.350	2.22	1.98	4.62	4.32

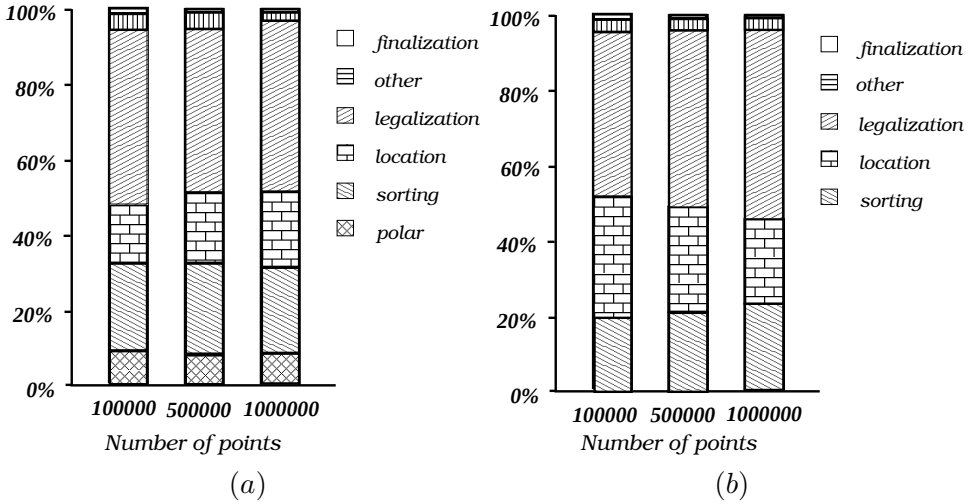

Figure 15. Typical run times needed for sweep-circle (a), and Žalik (b) algorithms.

and they are important for several applications such as FEM. Figure 16 displays the three examples of the data sets. According to the results which are shown in Table 4, the proposed algorithm is again the most efficient.

4.3 CPU time-independent comparisons

Firstly, we estimate the suitability of the heuristic introduced to detecting the basins. The main concern of this heuristic is to reduce the creation of non-Delaunay triangles and, as a consequence, to minimize the number of calls to the legalization procedure. We will show that the bottleneck of proposed algorithm is legalization. Table 5 gives the number of legalized triangles by the proposed algorithm, while the basin founded by the first and second heuristics. Sets of points containing 1 000 000 points were used. The second heuristic decreases the number of swaps around 4.5%.

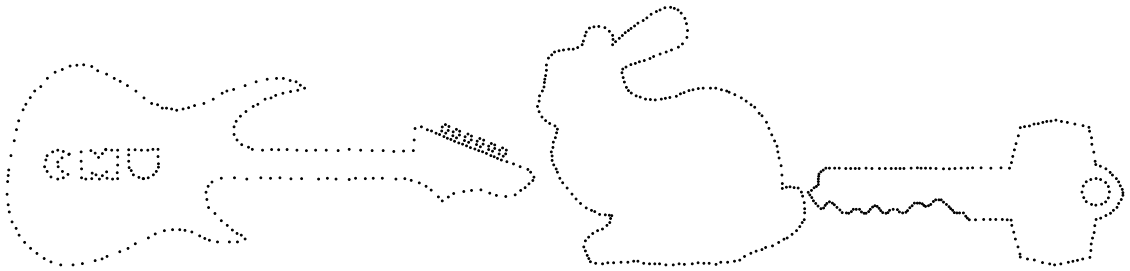

Figure 16. Points covering boundary of objects.

Table 4. Spent CPU time (s) for boundary point sets.

No. of points	CGAL	FS	GSD	ZS	SC
102 213	4.48	2.21	1.22	0.48	0.38
150 763	8.03	3.88	1.67	0.66	0.54
259 607	18.5	7	2.32	1.26	1.12
376 908	32	10.9	3.11	1.90	1.69
501 027	57	14.9	3.94	2.31	2.03

Table 5. Comparison of heuristics for detecting basins (the number of legalized triangles).

	No. of all triangles	First heuristic	Second heuristic
Uniform	1 999 912	629 966	602 052
Gaussian	1 967 729	604 970	571 251
Cluster	1 998 188	635 278	606 485
Grid-60°	1 996 002	192 695	191 411

We can measure the performance of frontier (hash table on circular list) by the number of table entries and list elements that must be searched to locate a new site. Table 6 compares the performance of the frontier in sweep-circle with the advancing-front and lower convex hull in Žalik, while triangulating 1 000 000 points. Sweep-circle performs around 1.5 table-searches per site for all sets. For points arranged in clusters the number of list-searches is around 13 point per site, because the frontier has no uniform distribution and all points of a cluster may be assigned to one entry of the hash-table and this can increase the number of list-searches for that entry. This number can be decreased by selecting smaller values for k, as in Žalik. It also can be decreased by using a hash table on balanced binary search tree(s) instead of linked list; but, the sweep-circle algorithm accesses to the predecessors and successors of frontier points frequently, which required $O(1)$ time on linked list and $O(\log p)$ on balanced binary search trees, where p is size of the tree assigned to an entry of the hash-table.

Table 6. Performance of the hash table and linked lists ($k = 100$ for Žalik, and $k = \sqrt[3]{n}$ for sweep-circle) , which defined as the number of searches per site.

	Žalik		Sweep Circle	
	table-search	list-search	table-search	list-search
Uniform	5.51	3.09	1.48	5.24
Gaussian	3.52	3.08	1.67	4.86
Cluster	1.35	3.76	1.15	12.86
Grid-60°	3.68	2.40	1.51	2.96

By deeper view on Tables 1, 2, 6 and Figure 15, we can find that the number of searches doesn't affect the run time considerably, because each test contains only one equality or non-equality comparison. Especially on cluster data sets that number of total searches is almost twice the number of searches on uniform and Gaussian data sets, but the triangulation time is fewer. Therefore, the main bottleneck in our algorithm and Žalik is updating the Delaunay diagram that contains *in-circle* tests and *edge flips* [4]. Su and Drysdale [37] also showed that the in-circle test is the most time consuming part of incremental algorithms. Žalik [39] compared his sweep-line algorithm with incremental insertion algorithm and showed that his algorithm requires only 26% of diagonal swaps regarding the incremental insertion algorithm. Here, we compare our sweep-circle algorithm with Žalik's sweep-line algorithm. Table 7 gives the number of in-circle tests and legalized edges per site using different

point distributions with 1 000 000 points. It also shows the spent CPU time for in-circle tests and legalizations. The proposed algorithm requires around 92% of diagonal swaps and 83% of CPU time regarding Žalik's sweep-line algorithm (see also Table 5).

Table 7. Number of in-circle tests and legalized edges per site, and their spent CPU time.

	No. of	Žalik			Sweep Circle			Flip	Time
	triangles	InC	Flip	Time	InC	Flip	Time	ratio	ratio
Uniform	1 999 912	4.56	0.650	2.35	4.30	0.602	1.87	0.93	0.80
Gaussian	1 967 729	4.50	0.634	2.30	4.19	0.571	1.75	0.90	0.76
Cluster	1 998 188	4.58	0.657	2.36	4.31	0.606	1.89	0.92	0.80
Grid-60°	1 996 002	3.38	0.193	1.60	3.37	0.191	1.55	0.99	0.97

Now, we compare the proposed method with the Dwyer's algorithm. Su and Drysdale [37] showed that the in-circle routine accounts for more time than any other routine in the algorithms (about half the time of both Dwyer and incremental algorithm). Table 8 compares the number of in-circle tests per site for 1 000 000 points; previously, the similar results about Dwyer also obtained in [37]. Dwyers algorithm spends most of its time for in-circle test. The number of in-circle tests in Dwyer is more than twice our algorithm.

Finally, we compare the proposed sweep-circle algorithm with Fortune's sweep-line algorithm. Both algorithms sort the input points at first. Fortune's algorithm adds Delaunay triangles to the diagram at circle events. In this way, legalization is not needed, but efficient maintenance of the frontier and event queue is demanding. Fortune's own implementation, uses bucketing for this purposes. Bucketing yields fast implementations on uniform point sets, but is easily defeated; a small, dense cluster of points in a large, sparsely populated region may all fall into a single bucket [35]. Su and Drysdale [37] found that the number of comparisons per site in event queue grows as $9.95 + 0.25\sqrt{n}$ on uniform random point sets, thus the Fortune's implementation exhibit $O(n\sqrt{n})$ performance. By re-implementing Fortune's code using an array-based heap instead of bucketing to represent priority queue, they obtained $O(n \log n)$ running time and better performance on large point sets. Shewchuk's [35] implementation in Triangle uses a heap to store event points, and a splay tree to store sweep-line status. Splay trees adjust themselves so that frequently accessed items are near the top of the tree. To test the effectiveness of Shewchuk's implementation, we performed an extensive experiment. The

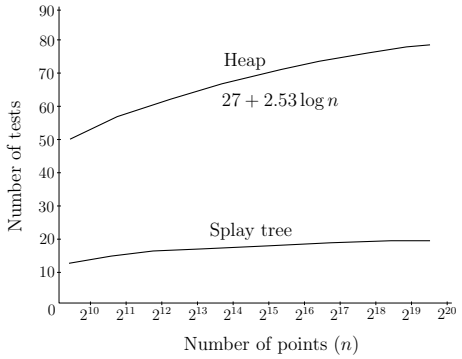

Figure 17. Cost of Fortune. Number of tests per site needed to maintain the heap and the splay tree.

algorithm was tested on uniform inputs with sizes ranging from 2^{10} to 2^{20} sites. Figure 17 shows the performance of the event queue and sweep-line data structures in the experiment. Regression analysis shows that the number of comparisons per site for maintaining the heap grows logarithmically as the

Table 8. Number of in-circle tests per site for sweep-circle and Dwyer.

	uniform	Gaussian	cluster	Grid-60°
SC	4.30	4.19	4.31	3.37
GSD	8.20	9.3	10.8	7.4

line $27 + 2.53 \log n$ in Figure 17.

The bottleneck of Fortune is the maintenance of event queue data structure, which is represented as an array-based heap. Event points are inserted into the heap according to their y-coordinates. There are n site events that known in advance and at most $2n - 5$ circle events [4], which have to be calculated during triangulation; giving us in total, at most, $3n - 5$ event points. Thus the sweep-line moves $O(n)$ times-once per site and once per triangle-and it costs $O(\log n)$ time per move to maintain the priority queue and sweep-line data structure. Therefore the running time of the algorithm is $O(n \log n)$. But, in sweep-circle algorithm, the frontier represented by a simple polygon, which is much easier to implement. It has only n known event points at the sites. Both algorithms are insensitive on the different distributions of the input points.

5 Conclusion

The experiments in this paper led to several important observations about the performance of the algorithms for constructing planar Delaunay triangulations. These observations are summarized below:

- This paper introduces a new $O(n \log n)$ expected time algorithm for constructing DT in the plane. The algorithm efficiently combines the sweep-circle paradigm with the legalization. It is easy to understand and very simple to implement and has a good performance.
- The bottleneck of Dwyer's divide-and-conquer, Žalik's sweep-line, and proposed sweep-circle algorithms is updating the Delaunay diagram which contains in-circle tests and edge flips. The proposed algorithm reduces the number of in-circle tests and edge flips. As a result, it is the fastest among all of them.
- In Shewchuk's implementation of Fortune's sweep-line, most of time spends for maintenance of the frontier and event queue, which costs $O(\log n)$ time per sweep-line move.

Acknowledgment

We would like to thank Prof. B. Žalik from the University of Maribor, Slovenia, for sharing his code with us and to J. R. Shewchuk for making his Triangle package publicly available. We would also like to thank Hamid Zarrabi-Zadeh suggestions, from University of Carleton, Canada.

References

[1] Adam B., Kauffmann P., Schmitt D. and Spehner J. C., An increasing-circle sweep-algorithm to construct the Delaunay diagram in the plane. Proceedings of CCCG, (1997)

[2] Barber C. B., Computational geometry with imprecise data and arithmetic. PhD thesis, Princeton (1993)

[3] Béchet E., Cuilliere J. C. and Trochu F., Generation of a finite element MESH from stereolithography (STL) files, Computer-Aided Design, 34, 1–17 (2002)

[4] Berg M. D., Kreveld M. van, Overmars M. and Schwarzkopf O., Computational geometry, algorithms and applications. Springer-Vetrlag (2008)

[5] Biniaz A., Circumcircular range searching in higher-order Delaunay triangulations. Proceedings of JCCGG (2009)

[6] Biniaz A., Slope preserving terrain simplification—an experimental study. Proc. of CCCG (2009)

[7] Biniaz A. and Dastghaybifard Gh., Drainage reality in terrains with higher order Delaunay triangulations. In Advances in 3D Geoinformation Systems, Springer-Verlag, 199–213 (2008)

[8] Biniaz A. and Dastghaybifard Gh., Slope fidelity in terrains with higher order Delaunay triangulations. Proceedings of WSCG (2008)

[9] Biniaz A. and Dastghaybifard Gh., A comparison of plane sweep Delaunay triangulation algorithms. Proceedings of CSICC (2007)

[10] Brown K. Q., Voronoi diagrams from convex hulls. Inf. Proc. Letters, 5, 223–228 (1979)

[11] CGAL. http://www.cgal.org, Version 3.1

[12] Cheng S. W. and Poon S. H., Three-Dimensional Delaunay Mesh Generation. Discrete and Computational Geometry, 36, 419–456 (2006)

[13] Cormen T. H., Leiserson C. E. and Rivest R. L., Introduction to Algorithms. MIT Press/McGraw-Hill, Cambridge, MA (1990)

[14] Dehne F. and Klein R., A sweep-circle algorithm for Voronoi diagrams (Extended Abstract). Lecture Notes in Comput. Sci., Vol. 314. Springer Verlag, Berlin (1988)

[15] Dwyer R. A., Higher-dimensional Voronoi diagrams in linear expected time. Discrete and Computational Geometry, 6, 343–367 (1991)

[16] Dwyer R. A., A faster divide-and-conquer algorithm for constructing Delaunay triangulations. Algorithmica, 2, 137–151 (1987)

[17] Fortune S., Voronoi diagrams and Delaunay triangulations. in: Goodman, J. E., O'Rourke, J.(eds.) Handbook of Discrete and Computational Geometry, Chapter 20, CRC Press LLC, Boca Raton, FL, 377–388 (1997)

[18] Fortune, S.: A sweep-line algorithm for Voronoi diagrams. Algorithmica, 2, 153–174 (1987)

[19] Gonçalves G., Julien P., Riazano S. and Cervelle B., Preserving cartographic quality in DTM interpolation from contour lines, ISPRS J. of Photogram. and Remote Sen., 56, 210-220 (2002)

[20] Gudmundsson J., Hammar M. and Kreveld M. van, Higher order delaunay triangulations. Computational Geometry: Theory & Applications, 23, 85–98 (2002)

[21] Gudmundsson J., Haverkort H. and Kreveld M. van, Constrained higher order delaunay triangulations. Computational Geometry: Theory & Applications, 30, 271-277 (2005)

[22] Guibas L., Knuth D. and Sharir M., Randomized incremental construction of Delaunay and Voronoi diagrams. Algorithmica, 7, 381–413 (1992)

[23] Guibas L. and Stolfi J., Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. ACM Transactions on Graphics, 4(2), 75–123 (1985)

[24] Kok T. de, Kreveld M. van and Löffer M., Generating realistic terrains with higher-order Delaunay triangulations. Computational Geometry: Theory and Applications, 36, 52-67 (2007)

[25] Kolingerova I. and Žalik B., Improvements to randomized incremental Delaunay insertion. Computer Graphics, 26, 477–490 (2001)

[26] Lawson C. L., Software for C^1 surface interpolation. In: Rice, J. R.(eds.) Mathematical software III, New York, Academic press, 161–194 (1997)

[27] Lee D. T. and Schachter B. J., Two algorithms for constructing a Delaunay triangulation. Int J Comput Inf Sci, 9, 219–42 (1980)

[28] Maus A., Delaunay triangulation and the convex hull of n points in expected linear time. BIT, 24, 151–163 (1984)

[29] Park J. H. and Park H. W., Fast view interpolation of stereo images using image gradient and disparity triangulation. Signal Processing: Image Communication, 18 , 401-416 (2003)

[30] Philip J., The Area of a Random Convex Polygon, Tech. Report TRITA MAT 04 MA 07

[31] Preparata F. P. and Shamos M. I., Computational geometry: an introduction. Springer, Berlin (1985)

[32] Shewchuk J. R., General-Dimensional Constrained Delaunay and Constrained Regular Triangulations, I: Combinatorial Properties. Discrete and Computational Geometry, 39, 580–637 (2008)

[33] Shewchuk J. R., A two-dimensional quality mesh generator, Delaunay triangulator. http://www.cs.berkeley.edu/jrs/index.html (2004)

[34] Shewchuk J. R., Lecture notes on Delaunay mesh generation. Univ. of Calif. at Berkeley (1999)

[35] Shewchuk J. R., Triangle: engineering a 2D quality mesh generator and Delaunay triangulator. Proceedings of SCG'96 (ACM Symposium on Computational Geometry), 124–133 (1996)

[36] Silva V. de, A weak characterisation of the Delaunay triangulation. Geometriae Dedicata, 135, 39-64 (2008)

[37] Su P. and Drysdale R. L. S., A comparison of sequential Delaunay triangulation algorithms. Proceedings of SCG'95 (ACM Symposium on Computational Geometry), 61–70 (1995)

[38] Tekalp A. M. and Ostermann J., Face and 2-D mesh animation in MPEG-4. Signal Processing: Image Communication 15, 387-421 (2000)

[39] Žalik B., An efficient sweep-line Delaunay triangulation algorithm. Computer-Aided Design, 37, 1027–1038 (2005)

[40] Žalik B. and Kolingerova I., An incremental construction algorithm for Delaunay triangulation using the nearest-point paradigm. Int. J. Geograph. Inf. Sci., 17, 119–138 (2003)