A Comparison of Plane Sweep Delaunay Triangulation Algorithms

Ahmad Biniaz[1], Gholamhossein Dastghaibyfard[2]

Department of Computer Science and Engineering, College of Engineering,
Shiraz University, Shiraz, Iran
[1] biniaz@cse.shirazu.ac.ir
[2] dstghaib@shirazu.ac.ir

Abstract

This paper presents a survey as well as a new sweep-circle algorithm, on plane sweep algorithms for computing the Delaunay triangulation. The algorithms examined are: Fortune's sweep-line algorithm, Zalik's sweep-line algorithm, and a sweep-circle algorithm proposed by Adam, Kauffmann, Schmitt, and Spehner.

We test implementations of these algorithms on a number of uniform and none-uniform distributed sites. 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.

Keywords

Delaunay triangulation, sweep line, sweep circle.

1. Introduction

Given a set *S* of *n* points in the plane called sites, the problem of triangulating *S*, that is, decomposing the convex hull of S in disjoint triangles whose vertices are the sites of *S*, has been studied extensively. Among these triangulations, Delaunay triangulations, *Del(S)* (whose triangles circumcircles contain no site in their interior) seem to be the most regular. Indeed, a Delaunay triangulation maximizes the minimum angle of its triangles, and moreover maximizes lexicographically the increasing sequence of these angles [11], which makes it convenient for different engineering applications. Several approaches have been proposed for its construction. Sequential algorithms for constructing it classified in to five groups [1]:

- divide and conquer algorithms [16, 17],
- sweep-line algorithms [2, 3],
- incremental algorithms [5, 13, 14, 15],
- gift-wrapping algorithms [9], and
- convex hull based algorithms [12]

This paper presents an experimental comparison of plane sweep algorithms. Implementations of these algorithms are tested on a number of uniformly distributed sites and a number of highly non-uniform distributions. In addition, we describe a new plane sweep algorithm, which is simple to understand and implement, it is competitive with the other. The algorithm uses a combination of sweep-circle paradigm and Lawson's [6] recursive local optimization procedure to achieve both simplicity and good performance.

The experiments in this paper go beyond measuring total run time, which is highly dependent on the computer system. 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.

This paper is organized as follows. Section 2 and 3 briefly describe the various sweep-line and sweep-circle algorithmic approaches. Section 4 gives experimental results and comparisons, and then, we conclude this paper in Section 5.

2. Sweep Line Algorithms

The sweep-line is one of the most popular acceleration techniques used to solve 2D geometric problems [4]. The idea of the sweep-line technique is very simple: firstly, the geometric elements are sorted. Then, it is imagined that the sweep-line glides over the plane and stops at so-called *event points* [2]. 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.

Remain of this section describes two sweep-line algorithms: Fortune [3] and Zalik [2]. In both of them it is assumed that the sweep-line moves in the *y* direction (upward).

2.1. Fortune's Sweep-line Algorithm

In 1987, Fortune [3] finds an $O(n \log n)$ scheme for applying the sweep-line approach for constructing Delaunay triangulation of *S*. The algorithm keeps track of two sets of states. The first is a sweep-line data structure that is an ordered list of sites called the *frontier* of the diagram. In addition, the algorithm uses a priority queue, called the *event queue* to keep track of next sweep-line move (place where the sweep-line should stop). This queue stores two types of events 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 a circle formed by three consecutive vertices on the frontier.

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

In original Fortune's implementation, the frontier is a simple linked list of half-edges, and point location is performed using a bucketing scheme according to the *x*-coordinate of a point to be located. A bucketing scheme is also used to represent the priority queue. Members of the queue are bucketed according to their priorities—their *y*-coordinates. In the version of sweep-line that implemented by Shewchuk in *Triangle* package [10]– the version that we use in this paper for experiments– 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. So, the next event can be accessed by extracting the minimum from heap.

2.2. Zalik's Sweep-line Algorithm

Zalik [2] proposed a fast $O(n \log n)$ expected time algorithm for constructing 2D Delaunay triangulation. It is based on sweep-line paradigm, which is combined with Lawson's [6] recursive local optimization procedure. The algorithm surrounds the swept vertices by two bordering polylines. The first is the upper polyline that called *advancing front*. The advancing front shows the state of sweep-line and separates the swept vertices from the non-swept. The second is a 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. Each vertex of the advancing front (or the lower convex hull) stores the pointer to the farthest right (the farthest left) triangle being defined by that vertex. The sweep-line stops only at sites and sites are sorted regarding their *y*-coordinates.

When the sweep-line meets the next event (site), a vertical projection of that site is done on the advancing front. According to the hit or miss of advancing front, four possible cases (HIT, ON-EDGE, LEFT and RIGHT) may appear. Fig. 1.a shows the most common case, when projection hits the advancing front. With three sites v_i, v_R and v_L, a new triangle is constructed and checked with neighboring triangle according to the empty circle property (Fig. 1.b). Two heuristics used to prevent the construction of tiny triangles, which would probably be *legalized* [5]: walking to the left and to the right on the advancing front, and removing *basins* [2]. The algorithm walks to the left and right on the advancing front and new triangles are constructed and legalized while the angle between three consecutive points is smaller than $\pi/2$ (Figs. 1.b and 1.c). In this way, the basin may appear and two monotone chains of vertices are constructed and triangulated by monotone polygon triangulation [5].

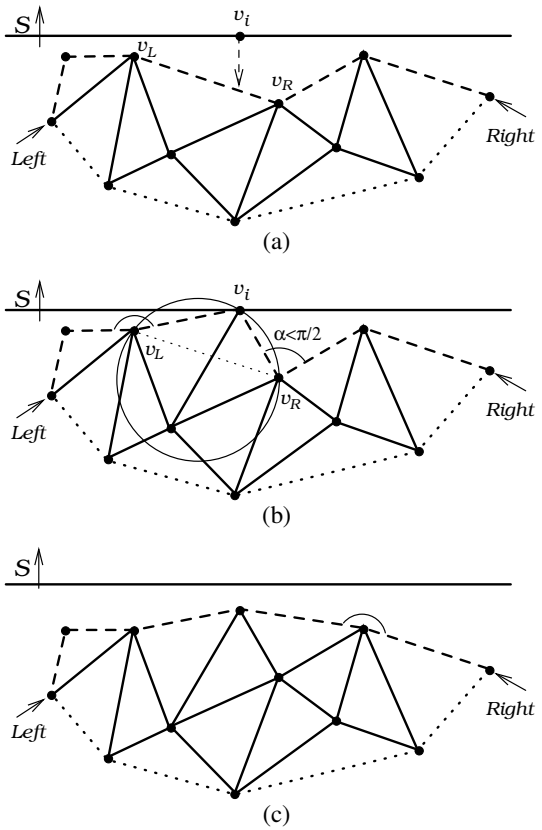

Fig. 1: The advancing front edges in dashed lines and lover convex hull edges in dotted lines: vertical projection of next site hits the advancing front (a), new triangle is constructed and legalized, advancing front is updated, walk to the right-side direction started (b), and stopped (c).

When all points are swept, some vertices of the frontier remain concave and some triangles are still missed. The missed triangles are added by performing a walk through the edges of the frontier.

3. Sweep Circle Algorithms

The sweep circle algorithms are based on the concept of wave front developed by Dehne and Klein [8]. These algorithms deal with the polar coordinates of sites. Let a point *O* in the convex hull of *S* be the origin of the polar coordinates. The idea of the sweep-circle technique is as follows: firstly, the geometric elements are sorted according to their distance from *O*. Then, it is imagined that the circle *C* whose center is *O*, increased and stops at event points. The part of the problem being swept (inside the circle) is already solved, while the remaining part (out of circle) is unsolved. As in sweep-line the problem is completely solved when the sweep circle passes through the last event point. This method is interesting notably when the Delaunay triangulation has to be constructed locally around a given point [7].

3.1. Adam's Sweep-circle Algorithm

Adam et. al. [7] introduced an algorithm that computes the Delaunay graph [5] by sweeping the plane with an increasing circle. The algorithm keeps track of a sweep-circle data structure that is an ordered list of sites called the *frontal* of the diagram. The algorithm also updates the current Delaunay graph by processing two kinds of events: *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 formed by three consecutive frontal sites, from *O* (Fig. 2). 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 contained no site of *S* in its interior.

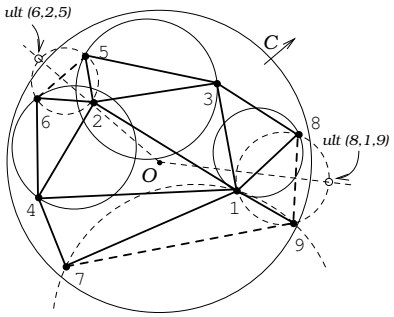

Fig. 2: The validated Delaunay edges in full-lines and the non-validated edges in dashed lines. The sites are numbered according to their distance from *O*. Front is (1, 9, 1, 7, 4, 6, 2, 5, 3, 8, 1). The ultimate points associated to frontal triples (6, 2, 5) and (8, 1, 9).

When *C* increases and sweeps over a site *s*, the algorithm searches a site(s) *t* in 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 tree when created and is deleted from tree when killed.

There are *n* site events. For every ultimate point swept, a region is updated, and since Delaunay graph of *S* admits at most $2n-4$ regions, the algorithm handles at most $3n-4$ events. Every search, insertion or deletion in balanced binary search tree is done in $O(\log n)$ time. Therefore the construction of Delaunay graph of *S* is done in $O(n \log n)$ time.

3.2. A New Sweep-circle Algorithm

We present an easily implemented new sweep-circle algorithm that is an extended version of Zalik's sweep-line algorithm [2]. It is based on sweep-circle paradigm, which is combined with Lawson's [6] recursive local optimization procedure. It performs better than algorithms mentioned above regardless of input sites distribution. The algorithm surrounds the triangulated vertices by a simple polygon that shows the sweep-circle status and called frontier. The frontier separates the swept vertices from non-swept. All vertices inside the frontier are triangulated regarding the Delaunay criterion. 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. Each vertex of the frontier stores the pointer to the farthest right triangle being defined by that vertex according clockwise direction.

Firstly, the origin of polar coordinates *O* must be selected such that be inside the polygon that formed by initial frontier edges, and then the polar coordinates of input points are calculated. The sweep-circle stops only at sites and sites are sorted according to their distance from *O* (their *r*-coordinates).

When the sweep-circle reaches the next unused event (vertex), the reaction of algorithm is as follows: The projection of new site toward *O* is done on the frontier. This projection always hits the frontier because *O* is inside the frontier and the new vertex is outside of it. Thus we have only HIT case. Fig. 3.a shows this only case. A new triangle is constructed and legalized if necessary. Two heuristics that are variation of heuristics mentioned in [2] are applied here. So, walking to the left and right on the frontier are started (Fig. 3.b shows the left-side walk) and stopped (Fig. 3.c) as in Zalik [2]. Basins are detected and removed. After sweeping all points, some triangles are still missed. Missed triangles are added to the diagram, by performing a walk on the frontier. The total expected runtime of this algorithm is also $O(n \log n)$.

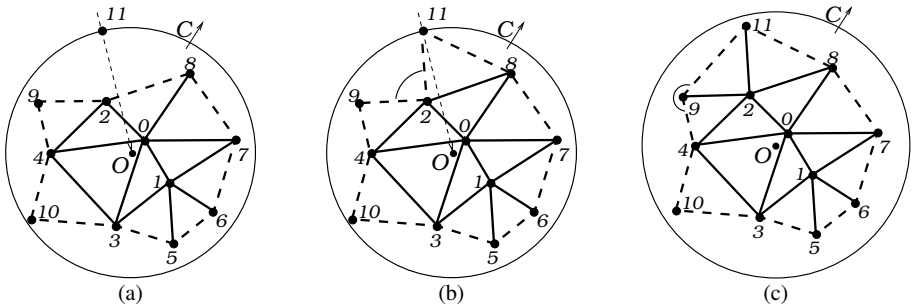

Fig. 3: The frontier edges in dashed lines. The sites are numbered regarding their *r* coordinates. projection of next site toward *O* hits the frontier (a), new triangle is constructed, frontier is updated, walking to the left side is started (b), and stopped (c).

4. Empirical Results

Adam et. al. [7] compared their sweep-circle algorithm with Fortune and showed that implementation of their algorithm is almost 20% slower than the sweep-line algorithm because the size of front in sweep-circle is bigger. Zalik [2] showed that his algorithm is the fastest among other popular Delaunay triangulation algorithms such as randomized incremental [13, 14, 15], divide & conquer [10, 16, 17] and Fortune's sweep-line [3]. Thus we only test the other three algorithms.

In order to evaluate the effectiveness of the algorithms described above, we studied C implementations of each algorithm. Steven Fortune's [3] sweep-line algorithm implemented by Shewchuk within a triangulation package Triangle [10], and Borut Zalik [2] provided code for his sweep-line algorithm. We implemented the sweep-circle algorithm. We test the algorithms on an ASUS A3000 lap top with Intel Pentium M+1.6 GHz processor and 504 MB of RAM running under the Microsoft Windows XP operating system.

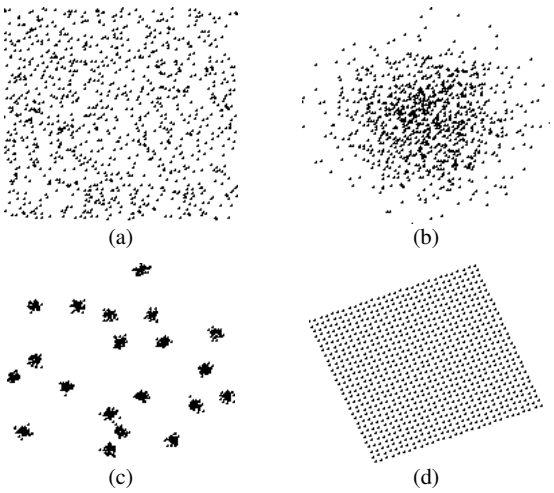

Fig. 4: Various point distributions: uniform (a), Gaussian (b), points arranged in clusters (c), and tilted grid (d).

Input points sets are used with uniform and Gaussian distributions and points arranged in clusters and tilted grid (Fig. 4). Table 1 compares the algorithms, including versions that use floating point computations. The proposed algorithm is fastest for all distributions, with the Zalik's algorithm second. Fortune's algorithm performs poorly, spending most of its time in searching event queue. For our algorithm, points arranged in regular grid represent the best case and points arranged in clusters represent the worst case.

4.1. Performance of Fortune

The run time of Fortune is proportional to the cost of searching and updating the event queue and sweep-line data structures. Fortune's own implementation, uses bucketing for this purposes. Su and Drysdale [1] found that the number of comparisons per site in event queue grows as $9.95+0.25\sqrt{n}$ on uniform random point set, 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 [10] implementation in Triangle uses a heap to store events and a splay tree to store sweep-line status.

The bottleneck of Fortune is the maintenance of event queue data structure, which is represented as an array-based heap. Events are inserted in to heap according to their *y*-coordinate. Only *n* site events are known in advance, the circle-events have to be calculated during triangulation.

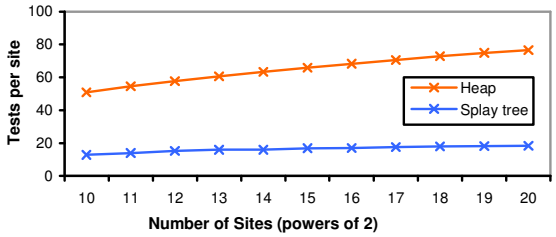

Fig. 5: Cost of Fortune. Number of tests per site needed to maintain the heap and splay tree.

To test the effectiveness of Shewchuk's implementation, we performed an extensive experiment. The algorithm was tested on uniform inputs with sizes ranging from 2^{10} to 2^{20} sites. Fig. 5 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 in the heap grows logarithmically as $27+2.53\lg n$ (orange line in Fig. 5), rather than as $\sqrt{n}$.

There are *n* point events and at most $2n-5$ circle events [5], giving us in total, at most, $3n-5$ events. 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 $O(n\log n)$.

Table 1: Timings for triangulation, not including I/O. Input points chosen from one of four distributions: uniformly distributed random points in a square, Gaussian points, points arranged in cluster, and points arranged in tilted grid.

No. of points	100,000				500,000				1,000,000			
Distribution	Unif.	Gaus.	Clus.	Grid	Unif.	Gaus.	Clus.	Grid	Unif.	Gaus.	Clus.	Grid
Fortune	1.16	1.12	1.06	1.13	6.95	6.58	6.07	6.81	15.1	14.3	12.9	14.7
Zalik	0.23	0.23	0.22	0.19	1.20	1.23	1.19	0.96	2.46	2.48	2.42	1.86
Sweep circle	0.20	0.20	0.22	0.19	1.07	1.00	1.12	0.94	2.18	2.15	2.27	1.83

4.2. Sweep-circle versus Zalik

In this subsection we compare the proposed sweep-circle algorithm with Zalik's sweep-line algorithm. Thus the word "algorithms" in this subsection refers to these two algorithms.

The run time of sweep-circle [resp. Zalik] is proportional to the cost of searching and updating the data structure representing the state of the sweep-circle [resp. sweep-line], and the cost of updating diagram when a new triangle is created. We use a hash-table on a circular double linked list for sweep-circle status. Zalik's implementation of sweep-line uses a hash-table on a double linked list for advancing front and a double linked list for lower convex hull.

When the sweep-circle [resp. sweep-line] sweeps an input point, the point must be located to the frontier [resp. advancing front] edges; this point location involves searching in hash-table for the first non-NULL entry followed by a search in a portion of double linked list that assigned to that entry. We called these two types of search, *table-search* and *list-search*, respectively. The size of frontier in proposed algorithm is bigger than the size of advancing front in sweep-line. This can be explained by the fact that the number of vertices of the convex hull of a set of n sites uniformly distributed in a circle is in $O(\sqrt[3]{n})$ but this number is in $O(\log n)$ for sites uniformly distributed in a rectangle [7]. Thus, using other data structures such as a heap or search trees–as in [10] or [7], for representing the frontier, increase the depth of locate procedure. By using bucketing scheme or a hash-table with an appropriate value for parameter k, the number of tests can be decreased. In these algorithms the value of k is determined by $1+\lfloor n/k \rfloor$.

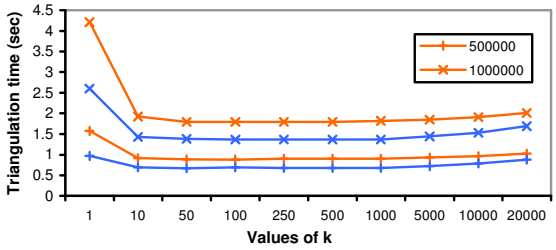

Fig. 6: Influence of parameter k on spent CPU time (without sorting), while triangulating 500,000 and 1,000,000 point sets. Orange lines for Zalik, and blue lines for sweep-circle.

Fig. 6 compares the spent CPU time in sweep-circle and Zalik, which is related to resolution of the hash-table, while triangulating different set of points at uniform distribution. Any value between 10 and 1000 for k gives very acceptable results. Worse results are obtained only if k is close to 1 (or too large). 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 [2]. On the other hand if k is too large, then there are smaller entries into hash-table, many sites are assigned for each entry of hash-table. This increases the spent time for list-search. In our experiments, $k = 100$ is used for Zalik and $k = 350$ is used for sweep-circle.

Table 1 shows the spent CPU time of algorithms. For points arranged in grid, these algorithms represent the best case.

Table 2 compares the performance of data structures used for sweep-line and sweep-circle status, while triangulating 1,000,000 points at different distributions. In our algorithm the search procedure performs around 1.5 table-searches per site, on average, while in Zalik this is around 3 for list-searches. For points arranged in clusters, Zalik represent the second best case, while sweep-circle represents the worse case. This is because that one cluster may be assigned to an entry of hash table and this can decrease the number of table-search in sweep-line and increase the number of list-searches in sweep-circle, while a new site assigned to that entry of hash table. If we use a hash table on balanced binary search tree(s) instead of linked list, the number of list-searches is decreased. But in these algorithms we access to the predecessors and successors of sites frequently, which is done in $O(1)$ on linked list and in $O(\log p)$ on balanced binary search trees–where p is the number of points assigned to an entry of hash table.

Table 2: Performance of hash tables and linked lists.

	Zalik		Sweep Circle	
	table	list	table	list
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° tilted)	3.68	2.40	1.51	2.96

By deeper view on Tables 1, and 2, 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 for sweep-circle on cluster data sets, that total number of searches is almost twice the number of searches on uniform and Gaussian data sets, but the triangulation time is almost the same. Therefore, the main bottleneck in these algorithms is updating the Delaunay diagram that contains *in-circle* tests and *edge flips* [5]. Edge flipping is a characteristic of incremental insertion algorithms. Zalik 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. Thus, we compare sweep-circle with Zalik from this aspect.

Table 3 gives the number of in-circle tests and legalized edges per site, with their total CPU time that done by algorithms using different point distributions. Each set contains 1,000,000 points. The proposed algorithm requires around 93% of diagonal swaps and 84% of in-circle and swaps time regarding Zalik's sweep-line algorithm.

Table 3: Number of in-circle tests and legalized edges per site, and their CPU time.

	No. of triangles	Zalik			Sweep Circle			Flip Ratio	Time Ratio
		In-C.	Flip	Time	In-C.	Flip	Time		
Uniform	1,999,912	4.56	0.650	1.08	4.30	0.602	0.91	0.93	0.84
Gaussian	1,967,729	4.50	0.634	1.04	4.19	0.571	0.85	0.90	0.82
Cluster	1,998,188	4.58	0.657	1.20	4.31	0.606	0.94	0.92	0.78
Grid (tilted)	1,996,002	3.38	0.193	0.67	3.37	0.191	0.62	0.99	0.93

Table 4 compares the spent CPU time on initialization and on the Delaunay triangulation phase of algorithms. Initialization in Zalik involves sorting input points by quick sort, but in sweep- circle it involves calculating polar coordinates of input points, followed by sorting them using quick sort. In Zalik 1/4 of total time is required for sorting, but in sweep-circle 1/3 of total time is required for polar coordinates calculation and sorting.

Table 4: Spent CPU time within Zalik and sweep-circle when triangulating point sets in uniform distributions.

No. of points	100,000		500,000		1,000,000	
Algorithm	Zalik	S.-C.	Zalik	S.-C.	Zalik	S.-C.
Initialization	0.05	0.06	0.31	0.40	0.69	0.81
Triangulation	0.17	0.14	0.89	0.67	1.77	1.37
Total	0.23	0.20	1.20	1.07	2.46	2.18

In sweep-circle, a waste of time is spent for calculating polar coordinates of input points. Thus, if we have the polar coordinates of them in advance, the running time of algorithm decreased considerably.

5. Conclusion

The experiments in this paper led to several important observations about the performance of plane sweep algorithms for constructing planar Delaunay triangulations. This paper introduces a new $O(n \log n)$ expected time algorithm for constructing Delaunay triangulation in the plane. The algorithm efficiently combines the sweep-circle paradigm with the legalization. It is easy to understand and very simple to implement. The bottleneck of this algorithm and Zalik's algorithm is updating the Delaunay diagram that contains in-circle tests and edge flips. By reducing these, the proposed algorithm, is the fastest plane sweep algorithm among others. In Shewchuk's implementation of sweep-line, the maintenance of frontier and the event queue costs $O(\log n)$ time per sweep-line move. This causes his heap based event queue implementation to perform better than Fortune's bucketing scheme on large inputs.

Acknowledgment

We would like to thank Prof. B. Zalik 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 Ph.d student, Hamid Zarrabi-Zadeh suggestions, from University of Waterloo, Canada.

References

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

[2] B. Zalik, "An efficient sweep-line Delaunay triangulation algorithm". Computer-Aided Design, 37:1027-1038, 2005.

[3] S. Fortune. "A sweep-line algorithm for voronoi diagrams". Algorithmica, 2:153-174, 1987.

[4] F. P. Preparata, M. I. Shamos. "Computational geometry: an introduction". Berlin, Springer, 1985.

[5] M. D. Berg, M. V. Kreveld, M. Overmars, O. Schwarzkopf. "Computational geometry, algorithms and applications". Berlin, Springer, 2000.

[6] C. L. Lawson. In: J. R. Rice, editor. "Software for C^1 surface interpolation". Mathematical software III. New York: Academic press, P. 161-194, 1977.

[7] B. Adam, P. Kauffmann, D. Schmitt, J.-C. Spehner. "An increasing-circle sweep-algorithm to construct the Delaunay diagram in the plane". Proceedings of CCCG '97 (Canadian Conference on Computational Geometry), 1997.

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

[9] R. A. Dwyer. "Higher-dimensional voronoi diagrams in linear expected time". Discrete & Computational Geometry, 6:343–367, 1991.

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

[11] D. Schmitt, J.-C. Spehner. "Angular Properties of Delaunay Diagrams in Any Dimension". Discrete and Comput Geom, Springer Verlag, New York Inc., 21:17-36, 1999.

[12] K. Q. Brown. "Voronoi diagrams from convex hulls". Information Processing Letters, 5:223–228, 1979.

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

[14] I. Kolingerova, B. Zalik. "Improvements to randomized incremental Delaunay insertion". Comput Graph, 26:477–90, 2001.

[15] B. Zalik, I. Kolingerova. "An incremental construction algorithm for Delaunay triangulation using the nearest-point paradigm". Int J Geograph Inf Sci, 17:119–38, 2003.

[16] D. T. Lee, B. J. Schachter. "Two algorithms for constructing a Delaunay triangulation". Int J Comput Inf Sci, 9:219–42, 1980.

[17] R. A. Dwyer. "A faster divide-and-conquer algorithm for constructing Delaunay triangulations". Algorithmica, 2:137–151, 1987.