アルゴリズム 27－6
（1992．5．21）

# 点位置決定アルゴリズムの実際的評価

加賀美 聡* 浅野 孝夫†

* 東京大学大学院工学系研究科 † 中央大学理工学部

『点位置決定問題』とは，「平面を直線分で分割した重なりのない領域（頂点数 $n$）が与えられたとき，質問点 $Q$ を含む領域を求める」という計算幾何学の中でも基本的な問題である．これまで点位置決定問題に対して種々のアルゴリズムが提案され，理論的に最適なもの（探索時間 $O(\log n)$，領域 $O(n)$，前処理時間 $O(n \log n)$）も幾つか知られている．

本論文では幾つかのアルゴリズムをC言語で作成し，頂点数 $2^{10} - 2^{17}$ の Voronoi 図を用いて計算機実験を行いその実用性を検討する．

# Practical Efficiencies of Planar Point Location Algorithms

Satoshi KAGAMI* and Takao ASANO†

* Graduate School of Engineering, University of Tokyo
Hongo, Bunkyo-ku, Tokyo 113, Japan
† Department of Information and System Engineering, Chuo University
Kasuga, Bunkyo-ku, Tokyo 112, Japan

The planar point location problem is one of the most fundamental problems in computational geometry, and stated as follows: Given a straight line planar graph (subdivision) with $n$ vertices and an arbitrary query point $Q$, determine which region contains $Q$. Many algorithms have been proposed, and some of them are known to be theoretically optimal ( $O(\log n)$ search time, $O(n)$ space and $O(n \log n)$ preprocessing time). In this paper, we implement several representative algorithms in C, and investigate their practical efficiencies by computational experiments on Voronoi diagrams with $2^{10} - 2^{17}$ vertices.

# 1 Introduction

The point location problem is one of the most fundamental problems in computational geometry, and stated as follows: Given a straight-line planar graph $G$ with $n$ vertices and a query point $Q$, determine which region of $G$ contains $Q$ (Fig. 1). An algorithm for this problem is generally characterized in terms of the following three attributes:

- **Preprocessing time** – the time required to construct a search structure.
- **Space** – the storage used for constructing and representing the search structure.
- **Search time** – the time required to locate a query point.

The first efficient algorithm was proposed by Dobkin and Lipton [2]. Their algorithm has $O(\log n)$ search time, $O(n^2)$ space, and $O(n^2 \log n)$ preprocessing time. Since then, several algorithms have been proposed, and some are optimal with respect to the worst-case complexity [5, 6, 8, 11]. Table 1 shows the theoretical efficiencies of representative algorithms.

From the practical point of view, Edahiro et al. [3] proposed an algorithm based on bucketing techniques and investigated its practical efficiency by comparing with the efficiencies of several then known algorithms (Kirkpatrick [6], Lee-Preparata [7], and Preparata [10] ). They claimed that their bucket algorithm was practically the most efficient among the tested algorithms. Since then, however, two new theoretically optimal algorithms have been proposed: one is by Edelsbrunner, Guibas and Stolfi [5] improving Lee-Preparata's algorithm, and the other is by Sarnak-Tarjan [11] using persistent search tree. Thus, it might be interesting to examine practical efficiencies of these two new optimal algorithms.

In this paper, we examine the algorithms of Edelsbrunner et al. [5] and Sarnak-Tarjan [11], by comparing with Bucket Method [3] and Lee-Preparata [7]. We implement these four algorithms in C, and investigate their practical efficiencies by computational experiments on Voronoi diagrams with $2^{10} - 2^{17}$ vertices. By combining this result with that of Edahiro et al. [3], we can compare all algorithms in Table 1 except Dobkin-Lipton [2], Shamos [12] and Lipton-Tarjan [8]. Note that, Dobkin-Lipton was improved by Shamos, and Shamos was improved by Preparata [3]. As to Lipton-Tarjan, it is extremely complicated and far from practical. Therefore, we believe these three algorithms could be omitted.

Table 1. Theoretical efficiencies of the representative algorithms

| Algorithm | Search | Space | Prep. |
|---|---|---|---|
| Dobkin-Lipton [2] | $\log n$ | $n^2$ | $n^2 \log n$ |
| Shamos [12] | $\log n$ | $n^2$ | $n^2$ |
| Lee-Preparata [7] | $(\log n)^2$ | $n$ | $n \log n$ |
| Lipton-Tarjan [8] | $\log n$ | $n$ | $n \log n$ |
| Preparata [10] | $\log n$ | $n \log n$ | $n \log n$ |
| Kirkpatrick [6] | $\log n$ | $n$ | $n \log n$ |
| Edelsbrunner et al.[5] | $\log n$ | $n$ | $n \log n$ |
| Sarnak-Tarjan [11] | $\log n$ | $n$ | $n \log n$ |
| Bucket Method [3] | $n$ | $n\sqrt{n}$ | $n\sqrt{n}$ |
| – Average – | 1 | $n$ | $n$ |

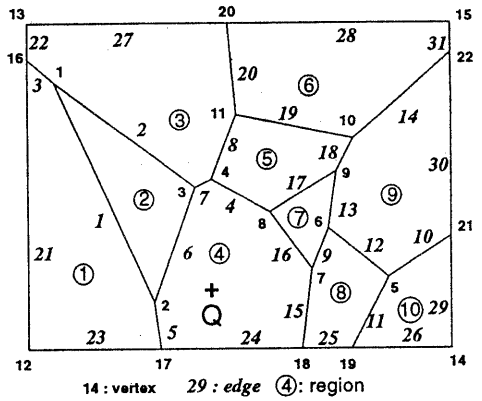



Fig. 1. Graph $G$ and a query point $Q$ ($Q$ is contained in region No.4)

# 2 Outlines of the Algorithms

In this section we outline the algorithms by Lee-Preparata [7], Edelsbrunner-Guibas-Stolfi [5], Sarnak-Tarjan [11], and Bucket Method [3].

Let $V = \{v_1, v_2, \cdots, v_n\}$ be the vertex set of a straight-line planar graph $G$ and $(x_i, y_i)$ a pair of coordinates of $v_i$ $(i = 1, 2, \cdots, n)$. Throughout this section we assume $y_1 \leq y_2 \leq \cdots \leq y_n$, that is, vertices are sorted in their $y$ coordinates by $O(n \log n)$ time preprocessing. All illustrations in this section are based on the graph in Fig. 1.

## 2.1 Lee-Preparata [7]

A *monotone chain* of the graph $G$ is a path from the lowermost vertex $v_1$ to the uppermost vertex $v_n$, each edge of which is directed upward. In this algorithm, the edges of the graph $G$ are decomposed into a set of monotone chains that are ordered by the left-right relation (Fig. 2(a)). $G$ can be decomposed in this way if and only if $G$ is *regular*, that is, every vertex is an initial point of an edge and a terminal point of another edge except the lowermost and uppermost vertices. Hence, if $G$ is not regular, $G$ is to be regularized by the plane-sweep algorithm in $O(n \log n)$ time in the preprocessing step.

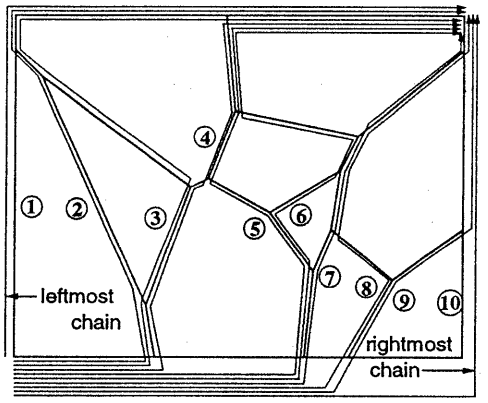
This algorithm locates a query point by finding a chain in the set that is adjacent to the point (a chain is *adjacent* to a point if it contains an edge incident to a region containing the point). Such a chain can be found by executing a sequence of binary searches among the set of chains. (Note that the chains are ordered by the left-right relation.) Each binary search solves a subproblem to determine which side of given monotone chain has the point. This can be done in $O(\log n)$ time. Furthermore, if the set of chains are stored in a balanced search tree, there are $O(\log n)$ binary searches in the sequence. Thus, the search time required by this algorithm is $O((\log n)^2)$.

However, if we store all edges of each chain, then space complexity will become $O(n^2)$. For $O(n)$ space complexity, we store each edge in only one chain (Fig. 2(b)). The following preprocessing constructs a search structure of $O(n)$ space supporting the above mentioned $O((\log n)^2)$ time search.

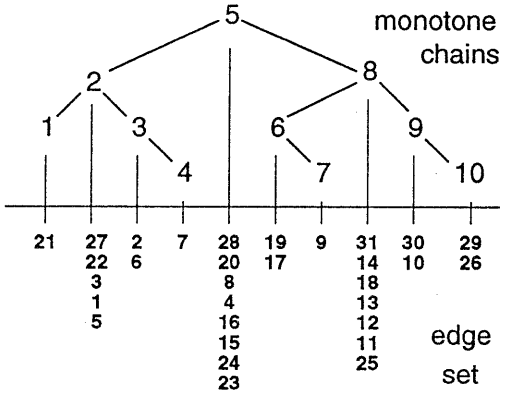
**Step 1.** Make $G$ regular (the resulting graph will also be denoted by $G$).

**Step 2.** Decompose $G$ into a set of monotone chains ordered by left-right relation. This can be done by assigning each edge $e$ of $G$ the leftmost chain $l(e)$ and the rightmost chain $r(e)$ containing $e$ (thus, exactly those chains from $l(e)$ to $r(e)$ contain $e$).

**Step 3.** Construct a balanced binary search tree $T$ in which each node $t$ is one-to-one corresponding to chain $c(t)$ in accordance with left-right relation. Each edge $e$ of $G$ is stored in exactly one node $t$ such that $c(t) \in [l(e), r(e)]$ and there is no ancestor $t'$ of $t$




(a) A decomposition of $G$ into a set of monotone chains




(b) Data structure

Fig. 2. Lee-Preparata

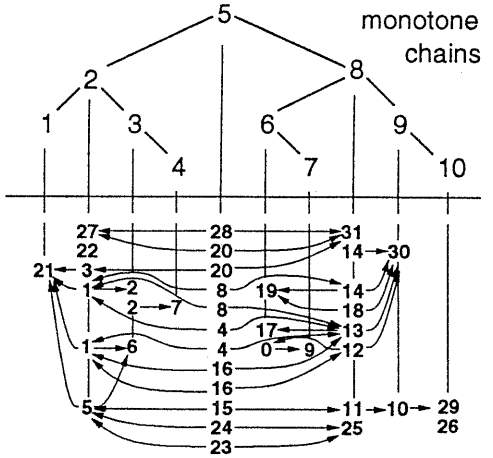



Fig. 3. Data structure for Edelsbrunner et al.

with $c(t') \in [l(e), r(e)]$. $E(t)$ denotes the set of edges stored in node $t$ of $T$.

**Step 4.** For each $E(t)$, construct a balanced binary search tree $T(t)$ of $E(t)$ based on the $y$ coordinates of edges.

Note that Steps 1 and 3 require $O(n \log n)$ time (Steps 2 and 4 require $O(n)$ time). Since each edge is stored in only one node, $E(t)$ does not exactly correspond to chain $c(t)$. $E(t)$ is a subset of $c(t)$, and consists of a set of subchains in $c(t)$ and gaps. We will not distinguish such a chain from a usual chain.

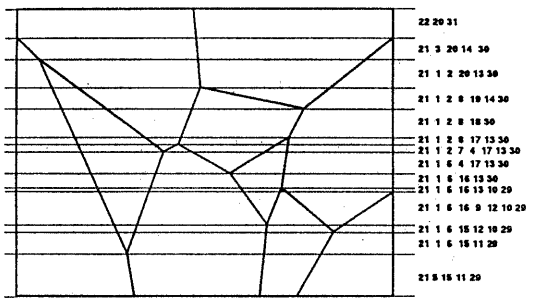


Fig. 4. Sarnak-Tarjan (A Slab decomposition)

## 2.2 Edelsbrunner-Guibas-Stolfi [5]

This algorithm is an improvement of Lee-Preparata, and attains $O(\log n)$ search time. From the chains stored in all the nodes of $T$ in Lee-Preparata, this algorithm obtains a new set of chains so that (1) once a point has been discriminated against a chain, it can be discriminated against a child of that chain with constant extra effort, and (2) the overall storage only doubles (so the space is $O(n)$).

Specifically, for each leaf $t$, we set $L(t) = E(t)$, and for each inner node $t$, we denote by $lc(t)$ its left child and by $rc(t)$ its right child. By traversing nodes $t$ of $T$ in post-order, new vertices are introduced as follows: add new vertices to $E(t)$ by choosing one from every two consecutive vertices in $L(lc(t))$ and $L(rc(t))$ and obtain $L(t)$. Thus an edge or gap in $L(t)$ can have a point in common with at most two edges or gaps in $L(lc(t))$ and in $L(rc(t))$, and (1) is attained. (2) is also attained which can be easily shown by an elemental calculation. This linked structure $L(t)$ is called a *layered dag* (Fig. 3). Thus the preprocessing to construct above mentioned search structure can be summarized as follows: (Steps 1 to 3 are the same as in Lee-Preparata)

**Step 4.** For each $E(t)$, construct $L(t)$, in post-order of nodes $t$ in $T$. For the root chain make a balanced binary search tree.

Note that Step 4 can be done in $O(n)$ time. With this structure, search is done as follows: "First find an edge that contains the $y$ coordinate of a query point $Q$ in the root chain by binary search, and determine in which side Q is. This can be done in $O(\log n)$ time. Next move to a left or right child chain appropriately, and find an edge in the current (child) chain that contains the $y$ coordinate of $Q$ by checking (at most two) edges having a point in common with the edge found in the previous (parent) chain and do this recursively until reaching a leaf. This can also be done in $O(\log n)$ time."

## 2.3 Sarnak-Tarjan [11]

This algorithm uses Slab Method proposed by Dobkin-Lipton [2] and Shamos [12]. Slab method is described as follows: "Draw a horizontal line through each vertex of the planar graph. This splits the plane into horizontal slabs. The edges of the graph intersecting a slab are totally ordered, from the left to the right of the slab. Associate with each edge its rightside region. Now it is possible to locate a query point with two binary searches: the first, on the $y$ coordinate, locates the slab containing the point; the second, on the edges intersecting the slab, locates the nearest edge lying to the left of the point, and hence determines the region containing the point." Sarnak-Tarjan's algorithm uses a persistent search tree as a data structure. A persistent search tree differs from an ordinary search tree in that after an insertion or deletion, the old version of the tree can still be accessed. The persistent search tree supports an update operation in $O(\log n)$ time and an access operation in $O(\log n)$ time, and requires $O(n)$ space.

For a planar graph, a persistent search tree is constructed based on the plane-sweep method: "A horizontal scan-line moving upwards stops at each vertex $v$ of the graph and an edge having $v$ as its upper endpoint is deleted from the persistent search tree and an edge having $v$ as its lower endpoint is inserted into the persistent search

tree." (Fig. 4)

As described above, search is : first find the slab containing a query point, and the second, find the edge lying just to the left of the query point. Thus, for a planar graph of $n$ vertices, the preprocessing time to build the data structure is $O(n \log n)$, the space is $O(n)$, and the search time is $O(\log n)$.

### 2.4 Bucket Method [3]

Consider a frame $R$ including a given planar graph $G$. For simplicity, the frame is restricted to a rectangle with sides parallel to the axis. The frame $R$ is partitioned into $n_x \times n_y$cells of equal size called *buckets* by $n_x - 1$ horizontal lines and $n_y - 1$ vertical lines. A bucket that is $i$th from the left and $j$th from the bottom is denoted by $B_{ij}$, and the subgraph of $G$ cut out by $B_{ij}$ is denoted by $G'_{ij}$ (Fig. 5).

In the preprocessing, construct data structures for respective buckets. The data structure for bucket $B_{ij}$ consists of $N_{ij}, H_{ij}, V_{ij}$, and $F_{ij}$ defined as follows:

- $N_{ij}$ : the list of vertices of $G'_{ij}$ sorted by their $y$ coordinates,
- $H_{ij}$ : the set of edges of $G'_{ij}$ intersecting the upper horizontal line of $B_{ij}$,
- $V_{ij}$ : the list of edges of $G'_{ij}$ having intersections with the left vertical line of $B_{ij}$ and sorted by the $y$ coordinates of intersections,
- $F_{ij}$ : a variable that indicates the region involving the upper left corner of $B_{ij}$.

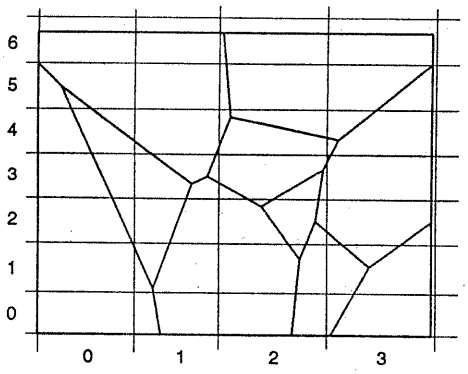
We can find a bucket $B_{ij}$ containing a query point in constant time. For the query point $Q$ in $B_{ij}$, compute

$$S = H_{ij} \oplus V_{ij}(Q) \oplus N_{ij}(Q) \oplus V_{i+1j}(Q).$$

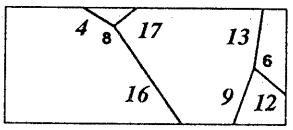
Here, $V_{ij}(Q)$ denotes the set of edges in $V_{ij}$ having intersections (with the left vertical line of $B_{ij}$) lying above $Q$. $N_{ij}(Q)$ denotes the set of vertices in $N_{ij}$ lying above $Q$. $A \oplus B$ means $(A - B) \cup (B - A)$. If $V_{ij}(Q) \neq \emptyset$ then set $f$ be the lower region of the lowest edge of $V_{ij}(Q)$, else set $f = F_{ij}$.

If the query point $Q$ lies to the left of the leftmost edge of $S$, we locate $Q$ in the region $f$. Otherwise, we locate $Q$ in the region lying to the right of rightmost edge among those in the left of $Q$.

This algorithm runs in time proportional to the size of the planar subgraph $G'_{ij}$, so that the search time is $O(1)$ in the average case if we choose $n_x$ and $n_y$ appropriately, and $O(n)$ in the worst case.




(a) Bucket partition of $G$




(b) $G'_{22}$ (bucket $B_{22}$)
($N_{22} = 8, 6.\ H_{22} = 4, 13, 17.\ V_{22} = \emptyset.\ F_{22} = 4.$)

Fig. 5. Bucket Method

## 3 Computational Experiments

In this section we investigate, through computational experiments, the practical efficiencies of the algorithms outlined in the previous section.

### 3.1 Design of Computational Experiments

We have examined the following algorithms in C on NEC EWS4800/220.

(1) Lee-Preparata [7],
(2) Edelsbrunner-Guibas-Stolfi [5],
(3) Sarnak-Tarjan [11],
(4) Bucket Method [3].

We tested the algorithms using the Voronoi diagrams with $2^{10} - 2^{17}$ vertices. The maximum size is $2^5$ times larger than that of Edahiro et al. [3]. We used the Voronoi diagrams whose generators are uniformly distributed in unit square[9]. Edahiro et al. [3] have examined algorithms using four types of input graphs including Voronoi

diagrams. However, there have been no distingushed differences among them. Therefore, here, we examine the algorithms using only Voronoi diagrams. The search time is measured as the average of those for query points.

### 3.2 Computational Results and Estimation of the Algorithms

We now show computational results of the experiments designed as outlined above, and evaluate the practical efficiencies of the implemented algorithms. Fig. 6 indicates search time, Fig. 7 indicates space, and Fig. 8 indicates preprocessing time. As is seen in the figures, it is observed that all the algorithms realize the asymptotic complexities shown in Table 1.

For search time, as is seen in Fig. 6, Bucket Method is the best, and Edelsbrunner et al. follows.

For space, as is seen in Fig. 7, Lee-Preparata is best, and Bucket Method is slightly worse than Lee-Preparata.

For preprocessing time, as is seen in Fig. 8, Bucket Method is the best. But it should be pointed out that the others are implemented with many subroutines for preprocessing step. Since Voronoi diagrams are regular, we do not have to regularize them, but we do so because of the comparison with other algorithms. In fact, almost all preprocessing time is consumed in the regularization step in Lee-Preparata and in Edelsbrunner et al. (Note that Lee-Preparata and Edelsbrunner et al. have to regularize the given graphs first.)

Now we estimate the efficiency of each algorithm. Lee-Preparata is the best for space, but it has a serious fault in $O((\log n)^2)$ search time. Edelsbrunner et al. is slightly worse than Lee-Preparata in preprocessing time, and needs twice as much space as Lee-Preparata, but search time is greatly improved. Compared with Edelsbrunner et al., Sarnak-Tarjan is slightly worse as to search time. As to space, it is worse. But as to perprocessing time it is better than Edelsbrunner et al. Except space, Bucket Method is the best. Its search time and preprocessing time attain $O(1)$ and $O(n)$, respectively.

### 3.3 Remarks

(1) Bucket Method is simple and its program is very short (about 350 steps), as compared with

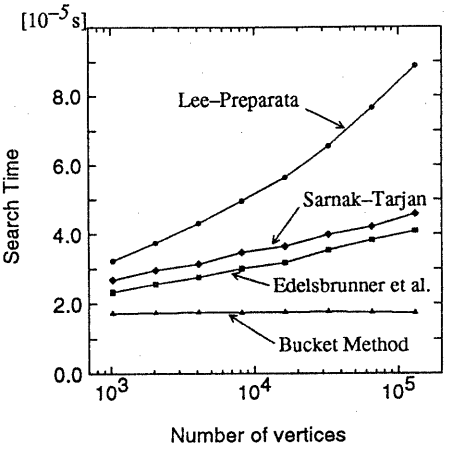


Fig. 6. Search time

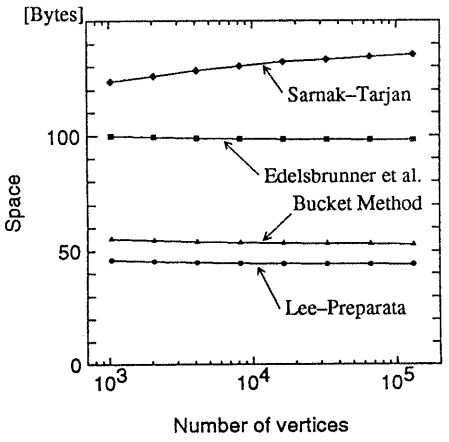


Fig. 7. Space (per vertex)

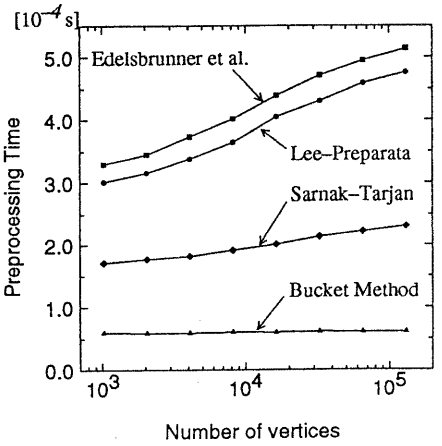


Fig. 8. Preprocessing time (per vertex)

Lee-Preparata (about 700 steps), Edelsbrunner et al. (about 1000 steps), and Sarnak-Tarjan (about 1000 steps).

**(2)** Since a given Voronoi diagram is generated by randomly distributed generators, the Voronoi diagram can be thought as a kind of a random plane graph, although all its regions are convex. Thus Bucket Method works in expected complexity. Other algorithms run in worst complexity shown in Table 1.

**(3)** The preprocessing times of Lee-Preparata, Edelsbrunner et al. and Sarnak-Tarjan contain $O(n \log n)$ sorting time to put vertices in order in their $y$ coordinates. Moreover, the preprocessing times of Lee-Preparata and Edelsbrunner et al. contain *regularization* time. As mentioned before, since Voronoi diagrams are regular, we do not have to regularize them, but we do so because of the comparison with other algorithms. Fig. 9 shows preprocessing time without regularization in these two algorithms. From this, we can observe that in preprocessing, Edelsbrunner et al. is slightly worse than Bucket Method and Lee-Preparata is almost the same as Bucket Method.

**(4)** Edahiro et al. [3] used *Heap Sort* for sorting the $y$ coordinates of vertices in the preprocessing, while we use *Quick Sort* here, to speed up three algorithms (Lee-Preparata, Edelsbrunner et al. and Sarnak-Tarjan). We also measured preprocessing time for these three algorithms using *Heap Sort*. However, *Heap Sort* was observed to take almost twice as much time as *Quick Sort*. Thus, Lee-Preparata with *Heap Sort* becomes almost always worst than Bucket Method even if regularization time is deleted. (Fig. 9.)

**(5)** Only bucket method needs space for search, because after receiving a query point $Q$, it makes search data structure in the bucket containing $Q$. Thus, space for search should be taken into account.

**(6)** We implement each algorithm by dividing it into few modules according to its function. But Bucket Method in Edahiro et al. [3] is implemented with only one module to get better performance. (They implemented other algorithms in Edahiro et al. [3] by dividing into modules.)

**(7)** The persistent search tree used in Sarnak-Tarjan [11] is a versatile data structure and has many applications. In fact, the point location is one application. Another application is the orthogonal segment intersection search problem: Given a set $S$ of $n$ horizontal line segments and a

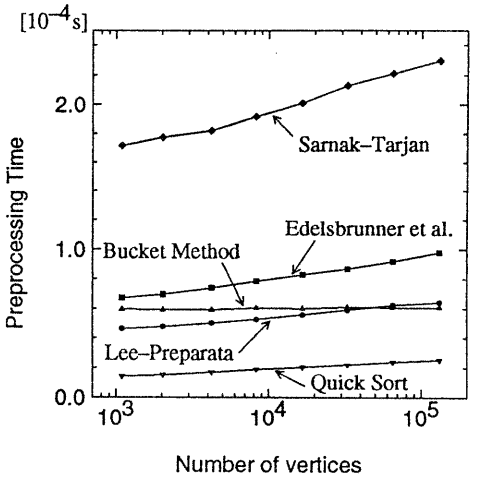



Fig. 9. Preprocessing time (per vertex) (Time for regularization is deleted from Lee-Preparata and Edelsbrunner et al.)

query vertical line segment $L$, find the set of segments in $S$ intersecting $L$. This problem can be solved optimally in $O(k+\log n)$ search time with $O(n)$ space and $O(\log n)$ preprocessing time, if the persistent search tree is used. Here $k$ is the number of segments in $S$ intersecting $L$. Another optimal algorithm was given by Chazelle [1]. Edahiro et al. [4] proposed an algorithm with buckets and implemented four representative algorithms in FORTRAN and investigated their practical efficiencies. However, Sarnak-Tarjan's algorithm was not examined. Thus, it might be interesting to examine its practical efficiency comparing with those of the algorithms examined in Edahiro et al. [4]. Persistent search tree used here in Sarnak-Tarjan is an improvement of ephemeral red-black tree. To make a tree persistent, we prepare five extra fields for each node: a third pointer called *slot* (left and right are first and second pointers), and times of left, right, slot and the node created. When attempting to add a node or change a pointer, if slot being used, copy the node, and do recursively to its parent. If slot is empty, assign left or right pointer appropriately to the slot. Amortized over a sequence of updates, only $O(1)$ nodes are copied per update, implying an $O(n)$ space bound for this persistent tree.

In the preprocessing step of Sarnak-Tarjan, to make search structure in each slab, when the

scan line makes a stop at the bottom point of the slab, we must delete only edges that come to this point, and insert edges that come out of this point. Since inserted edges will be in the same position, and pairs can be made between inserted edges and deleted edges, and, for each pair of an inserted edge and a deleted edge, copy the node containing the deleted edge (and thus the new node now containing the inserted edge), instead of doing delete and insert actually.

(8) In Bucket Method, the partition numbers $n_x$, $n_y$ are defined by using two positive parameters $\alpha_x$ and $\alpha_y$:

$$n_x = \max\{1, \lfloor \alpha_x \sqrt{n} \rfloor\},\ n_y = \max\{1, \lfloor \alpha_y \sqrt{n} \rfloor\}$$

Then the number of buckets becomes $O(n)$, and consequently, the number of vertices in a bucket becomes $O(1)$ on the average. But the number of edges in the bucket does not necessarily become $O(1)$, because bucket partition lines divide an edge into several segments. To make Bucket Method more efficient, $\alpha_x$ and $\alpha_y$ are chosen to satisfy:

$$\alpha_x : \alpha_y = S_x/x_d : S_y/y_d$$

where $S_x$ and $S_y$ are the total horizontal length and total vertical length of the segments respectively, and $x_d$ and $y_d$ are the horizontal width and vertical width of the given graph. We implemented the Bucket Method with $\alpha = \sqrt{\alpha_x \alpha_y} = \sqrt{e/n}$ ($e$ is the number of edges).

# 4 Conclusions

By combining the results in this paper and those in Edahiro et al. [3] we have the following conclusions. Bucket Method is the most efficient in many practical applications, although it has a weak point in the worst case complexity. Among the theoretically efficient algorithms including optimal ones, Edelsbrunner et al. and Sarnak-Tarjan are quite efficient as well as robust and thus work quite well in many practical applications. Especially, if the given graph is known to be regular in advance, then Edelsbrunner et al. is almost comparable with Bucket Method.

Thus, we would like to recommend to use Bucket Method in usual applications if given input graphs are not far from uniform, and Edelsbrunner et al. otherwise.


# Acknowledgements

The authors would like to express their gratitude to Prof. S. Suzuki, Dr. Y. Ishizuka and Dr. H. Yamashita of Sophia University. The second author was partly supported by the Grant-in-Aid for Scientific Research of the Ministry of Education, Science and Culture of Japan.


# References


[1] Chazelle, B.M. (1986): Filtering Search : A New Approach to Query-Answering. *SIAM J.Comput.*, Vol.15, pp.703–724.

[2] Dobkin, D. and Lipton, R.J. (1976) : Multidimensional Searching Problems. *SIAM J. Comput.*, Vol.5, pp.181–186.

[3] Edahiro, M., Kokubo, I. and Asano, T. (1984) : A New Point Location Algorithm and Its Practical Efficiency — Comparison with Existing Algorithms. *ACM Trans. Graphics*, Vol.3, pp.86–109.

[4] Edahiro, M., Tanaka, K., Hashino, T. and Asano, T. (1989) : A Bucketing Algorithm for the Orthogonal Segment Intersection Search Problem and Its Practical Efficiency. *Algorithmica*, Vol.4, pp61–76.

[5] Edelsbrunner, H., Guibas, L.J. and Stolfi, J. (1986) : Optimal Point Location in a Monotone Subdivision. *SIAM J. Comput.*, Vol.15, pp.317–340.

[6] Kirkpatrick, D.G. (1983) : Optimal Search in Planar Subdivisions. *SIAM J.Comput.*, Vol.12, pp.28–35.

[7] Lee, D.T. and Preparata, F.P. (1977) : Location of a Point in a Planar Subdivision and Its Applications. *SIAM J. Comput.*, Vol.6, pp.594–606.

[8] Lipton, R.J. and Tarjan, R.E. (1977) : Applications of Planar Separator Theorem. *Proc. 18th IEEE Symp.*, Foundations of Computer Science, IEEE New York, pp.162–170.

[9] Ohya, T. (1983) : On Efficient Algorithms for the Voronoi Diagram (in Japanese). Master's Thesis, Dept. of Math. Engineering, Univ. of Tokyo.

[10] Preparata, F.P. (1981) : A New Approach to Planar Point Location. *SIAM J.Comput.*, Vol.10, pp.473–483.

[11] Sarnak, N. and Tarjan, R.E. (1986) : Planar Point Location Using Persistent-Search Trees. *Communications of the ACM*, Vol.29, No.7, pp.669–679.

[12] Shamos, M.I. (1975) : Geometric Complexity. *Proc. 7th ACM Symp. Theory of Computing*, ACM, New York, pp.224–233.