Three Major Extensions to Kirkpatrick's Point Location Algorithm

A. Z. B. Haji Talib
School of Computer Sciences
University of Science Malaysia
11800 USM Penang, Malaysia
azht@cs.usm.my

M. Chen and P. Townsend
Department of Computer Science
University of Wales Swansea
Swansea SA2 8PP, United Kingdom
{M.Chen, P.Townsend}@swansea.ac.uk

Abstract

We present three extensions to Kirkpatrick's point location algorithm. The extensions revolve around major focuses in the point location problem namely the location of close successive query points and the dynamic and persistent implementations. The solutions described in this paper include (a) an algorithm that requires O(1) query time for reasonably close successive query points, with O(N) space bound and O(N) time to construct the search structure; (b) a dynamic implementation that require O(N) space, O(log N) query time and O(log N) update time; (c) a persistent implementation of the same complexity as the dynamic implementation. Two methods for modifying the search structure, namely K-structure, dynamically and persistently are also discussed. The space requirements and query times of the dynamic and persistent methods are identical to their static counterparts. To affirm the practicality, conceptual simplicity and efficiency of the new methods, we implemented all these extensions, performed computational tests and compared them with other alternative schemes. The results reflect the expected efficiencies of the underlying design.

1. Introduction

In geometric searching, the point location problem [1~5] stands out as one of the fundamental primitives of computational geometry. In planar point location, given a subdivision S of the plane and a query point q, the problem is to determine which region R of S contains q. Since the appearance of the first optimal planar point location method by Dobkin and Lipton [6], significant and substantial progress has been made in the area. One of the best known planar point location methods are Kirkpatrick's method [7] that constructs O(N) space search structure in O(N) time and allows O(log N) point location queries. Another method by Edelsbrunner et al [8] achieves the same bounds by applying the bridge-chained method to Lee and Preparata's algorithm [9].

The research into the point location problem has in recent years been focused on the development of dynamic and persistent methods as well as spatial methods. A *dynamic implementation* [3, 5, 10~16] allows a subdivision to evolve through deletion and insertion of its components. Some of the best known dynamic planar point location methods for monotone subdivisions include the method by Preparata and Tamassia [10], the two methods by Cheng and Janardan [12], and the method by Chiang and Tamassia [13]. Other dynamic methods include the methods by Goodrich and Tamassia [15], by Tamassia [16], and by Baumgarten et al [17]. A *persistent implementation*, which is a very recent focal point of research in the area, allows access and queries to the previous versions of the subdivision as well as the present one. The problem was introduced by Cheng and Janardan [12] who developed a persistent extension to their dynamic method. Most of these methods are based on their static counterparts [1~5], including the separating chain method and the trapezoid method. No attempt has been made to extend the Kirkpatrick's method though it is usually regarded as one of most practical methods if not the most.

In this paper, we present several extensions based on the Kirkpatrick's point location algorithm, which revolve around the three research focuses in the point location problem. The first extension allows close successive query points to be located in O(1) query time, with O(N) space bound and O(N) time to construct the search structure. The solution is practically very useful in computer graphics and geographical information systems. No such solution has been reported for the point location problem in the literature.

For the dynamic and persistent implementations of Kirkpatrick's method, we researched into a number of different schemes [5]. In this paper, we describe the best solutions that require O(N) space, O(log N) query time and O(log N) update time. We also include two methods of maintaining a dynamic and persistent data structure. Both the space bounds and the query times of these two extensions are identical to their optimal static counterparts. The testing results have demonstrate the efficiency of the methods chosen for the two extensions.

It is widely acknowledged that the study of geometric problems as is common in the area of computational geometry contributes to the development of highly efficient methods for the manipulation of graphical data

0-8186-7518-7/96 $5.00 © 1996 IEEE

[18~20]. These extensions have filled a number of gaps in the point location problem and will contribute a great deal to the future advances in computer graphics and scientific visualisation. The reason that Kirkpatrick's method was chosen as a basis for this work is because it is more readily applicable to triangular subdivisions which are commonly found in computer graphics and scientific computation. It is believed that there are ample opportunities and tremendous potential for this work.

The rest of the paper is structured as follows. An overview of the Kirkpatrick's method is to be given in Section 2. Section 3 describes the first extension to Kirkpatrick's method for close successive query points. After a brief discussion on several possible schemes, a dynamic implementation is presented in Section 4, where two methods of modifying dynamic Kirkpatrick search structure are also discussed. A persistent implementation is presented in Section 5 based on the methods developed for the dynamic implementation. The paper concludes in Section 6 with some computational results, complexity analysis and remarks on the work presented.

2. Kirkpatrick's point location algorithm

The method of Kirkpatrick [1~5, 7] is based on the use of a hierarchical data structure created for a triangulated subdivision. Given a triangular subdivision **S** that is enclosed in a triangle and has N vertices in total, a sequence of triangulation $\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_{h(N)}$ can be constructed, where h(N) is the height of the subdivision hierarchy, $\mathbf{S}_1 = \mathbf{S}$ and $\mathbf{S}_i$ is obtained from $\mathbf{S}_{i-1}$ as follows:

1. *choose a set of independent internal vertices* $\{v_1, v_2 \ldots, v_k\}$ *in* S_{i-1}.
2. *remove* $\{v_1, v_2 \ldots, v_k\}$ *and their incident edges.*
3. ***for*** *each of* $v_i \in \{v_1, v_2 \ldots, v_k\}$ ***do***
- 3.1. *re-triangulate the polygon arising from the removal of* v_i.

A set of vertices are said to be *independent* if none of them are adjacent to each other. $\mathbf{S}_{h(N)}$ is a subdivision defined by a single triangle without any internal vertices. If $|\mathbf{S}_i|$ denotes the number of vertices of $\mathbf{S}_i$, we have $|\mathbf{S}_1|=N$, $|\mathbf{S}_{h(N)}|=3$ and usually $|\mathbf{S}_{i-1}| >> |\mathbf{S}_i|$. A hierarchical data structure, which is depicted as an acyclic digraph in Figure 1, can be constructed where nodes represent all triangles of $\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_{h(N)}$ and each edge indicates the intersection between two triangles in $\mathbf{S}_i$ and $\mathbf{S}_{i-1}$ respectively. Such a process is called *Kirkpatrick decomposition*, and the resulting data structure is referred to as *K-structure*, denoted as $\mathcal{K}(\mathbf{S})$. In Figure 1, the edge shared by triangles T1 and T4 and that shared by T2 and T3 are joined together with first degree continuity. This condition is not allowed in Kirkpatrick's original algorithm, but it is permitted in our implementation for flexibility and practicality.

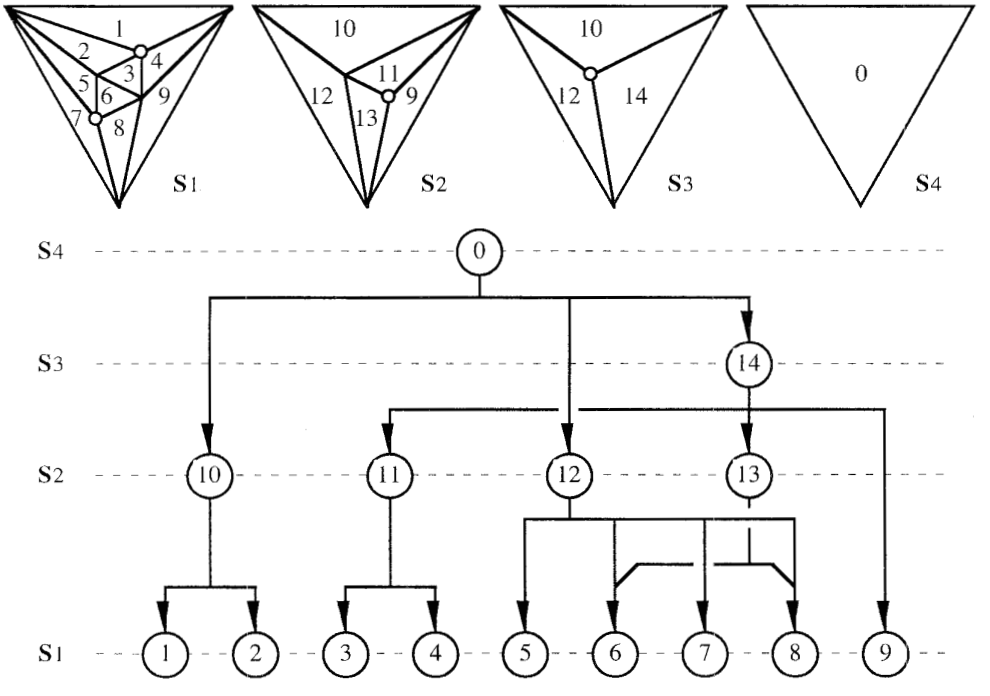

Figure 1. An example of a sequence of triangulation and the corresponding acyclic digraph.

The point location algorithm involves a search through $\mathcal{K}(\mathbf{S})$ locating a triangle that contains the query point $\mathbf{q}$ at each level, and employs an O(1) *point-in-triangle* test. The method achieves an O(log N) query time and a space bound of O(N). For pre-processing, O(N log N) time is required for the triangulation of $\mathbf{S}$, O(N) for the inclusion of the $\mathbf{S}$ in a triangle, and for the construction of $\mathcal{K}(\mathbf{S})$.

3. Point location for close successive query points

Many applications require an algorithm that allows repetitive queries can be processed efficiently. The aim in this case is to achieve a constant query time for locating successive points that are reasonably close to each other. An obvious strategy is to test the inclusion of the new point in the region $\mathbf{R}$ where the previous point was located. Failing this, the neighbouring regions are then examined. In effect, the main requirement is to formulate a systematic way of navigating around the regions of a subdivision with a given point.

Such a strategy can be applied to many existing point location methods [5]. However, most methods (such as the trapezoid method) treat the x and y directions differently, and a consistent navigation strategy can not be easily derived. This work identifies that Kirkpatrick's method is most suitable for the strategy outlined above, as the method does not give priority to either direction.

The extension involves the use a doubly-linked data structure for a $\mathcal{K}$-structure $\mathcal{K}(\mathbf{S})$ by adding a set of parent pointers into each triangle, allowing fast access to the neighbouring triangles through these parent triangles. The new data structure is denoted as $\mathcal{K}^{(2)}(\mathbf{S})$. An algorithm for navigating through $\mathcal{K}^{(2)}(\mathbf{S})$ has been implemented in a manner similar to the neighbour-finding algorithm for quadtree [21]. Given a set of query points $\mathbf{q}_1, \mathbf{q}_2, \ldots$, that are closely located, the search for the containment of $\mathbf{q}_i$ starts from the very same leaf triangle where $\mathbf{q}_{i-1}$ was located. (For $\mathbf{q}_1$ or when $\mathbf{q}_{i-1}$ is not inside any triangle, the search starts with the root triangle.) If this triangle does not contain $\mathbf{q}_i$, the search continues by examining the parent triangles through the use of parent pointers in $\mathcal{K}^{(2)}(\mathbf{S})$.

With $\mathcal{K}^{(2)}(\mathbf{S})$, a point location algorithm for arbitrarily distributed query points and one for close successive query points can both utilise the same data structure. The parent pointers can be assigned to each triangle during the process of Kirkpatrick decomposition without introducing additional complexity. A switching mechanism between two algorithms will be discussed in Section 6.1.

4. Dynamic point location

In many applications, the subdivision concerned is not static but requires dynamic changes regularly. It is necessary to maintain an appropriate data structure for the subdivision and its search structure, enabling update operations to be performed efficiently. We have therefore extended the Kirkpatrick's method to accommodate such a need, by designing and implementing a dynamic $\mathcal{K}$-structure, $\mathcal{K}^{(d)}(\mathbf{S})$.

4.1 Update operations

It is generally more challenge to implement a dynamic data structure for a triangular subdivision than a general subdivision, as the triangular nature has to be maintained. Partially because of this, the type of update operations allowed for a triangular subdivision is often restricted. Tamassia [16] proposed a set of three update operations in conjunction with his dynamic point location method. We herein-below propose a much more comprehensive set of operations:

- INSERT_VERTEX($\mathbf{v}$) — Inserting a vertex $\mathbf{v}$. There are two cases to be considered:
  a. Insert $\mathbf{v}$ into a triangle $\mathbf{t}$: Triangle $\mathbf{t}$ will be split into three new triangles (as in Tamassia's method [16]).
  b. Insert $\mathbf{v}$ on an edge $\mathbf{e}$: Each of the two triangles that share $\mathbf{e}$ will be split into two new triangles. Inserting $\mathbf{v}$ on a boundary edge affects only one triangle.
- DELETE_VERTEX($\mathbf{v}$) — Deleting a vertex $\mathbf{v}$. All the triangles and edges incident to $\mathbf{v}$ will be removed, and the resulting polygon will then be re-triangulated.
- MOVE_VERTEX($\mathbf{v}$, [x, y]) — Moving a vertex $\mathbf{v}$ to a new position [x, y]. Two cases are to be considered:
  a. If $\mathbf{v}$ does not cross any of edges, no re-triangulation is required.
  b. If $\mathbf{v}$ crosses at least one edge (which can be detected by testing the orientation of the triangles incident to $\mathbf{v}$), use operation DELETE_VERTEX($\mathbf{v}$) followed by INSERT_VERTEX([x, y]).
- INSERT_EDGE($\mathbf{e}$) — Inserting an edge $\mathbf{e}$ that connects two existing vertices. Two different operations are to be considered:
  a. The edges that intersect $\mathbf{e}$ are *not removed*: Edge $\mathbf{e}$ will split the intersecting triangles into two polygons. In most cases, a new edge has to be introduced to triangulate each of the resulting polygons.
  b. The edges that intersect $\mathbf{e}$ are *removed*: The insertion results in two polygons separated by $\mathbf{e}$. The two polygons have to be re-triangulated.

- DELETE_EDGE(**e**) — Deleting an edge **e**. There are two different operations to be considered, and the first operation has two cases:
  a. The endpoints are *not removed*, and a convex quadrangle is produced. The quadrangle will be divided into two triangles by adding a new edge other than **e** (as in Tamassia's method [16]).
     The endpoints are *not removed*, and a concave quadrilateral element is produced. This operation will revert to the previous triangulation, as any attempt for a new one would result in an invalid triangulation.
  b. The endpoints are *removed*, together with all other edges incident to the endpoints. The resulting polygon will be re-triangulated.

The above set of updates is adequate for most applications if not all. More complicated operations, such as moving an edge and inserting/deleting/moving a chain of edges (e.g. a triangle as a special case), can be performed by grouping several above operations together.

4.2 The Dynamic $\mathcal{K}$-structure — $\mathcal{K}^{(d)}(S)$

The most fundamental requirement of a $\mathcal{K}^{(d)}(S)$ is to support fast topological queries and to ensure efficient updates. In its original form [5, 7], the static $\mathcal{K}(S)$ has already facilitates one particular update operation namely removing a vertex and its incident edges during Kirkpatrick decomposition. From Section 4.1, one probably has already noticed that edge-based updates are generally more difficult to perform. This justifies the need for either an edge-based representation of $\mathcal{K}^{(d)}(S)$, or maintaining redundant edge information in a $\mathcal{K}^{(d)}(S)$. Furthermore, it is also desirable to have explicit information on triangles as suggested by many previous work on data structures for subdivision [22~26]. We adopted an edge-based representation for $\mathcal{K}^{(d)}(S)$ with two additional pointers for each edge. As each internal edge separates exactly two triangles, the two pointers are used to indicate the next edges of the two triangles, respectively, in clockwise order. For boundary edges, only one pointer is used. With such a $\mathcal{K}^{(d)}(S)$, topological queries on vertices, edges and triangles can all be answered quickly.

Figure 2 shows an example of inserting a vertex on an edge. Once edge **a** is located, the two triangles **Δabc** and **Δade** and related vertices can be found easily with the $\mathcal{K}^{(d)}(S)$. By adding three new edges **s**, **t** and $\mathbf{a}_2$ (and implicitly changing **a** to $\mathbf{a}_1$), the update operation is performed in O(1) time following the initial search for **a** in O(log N) time.

One possible approach to modification of a $\mathcal{K}^{(d)}(S)$ is simply to rebuild a new $\mathcal{K}(S)$ [27]. We have implemented such a method in order to make a comparison with a more sophisticated method outlined in Section 4.3.

4.3 Methods for modifying $\mathcal{K}^{(d)}(S)$

In almost all cases, the cost of simply modifying the current $\mathcal{K}^{(d)}(S)$ is much lower than rebuilding the whole $\mathcal{K}(S)$. Once all vertices, edges and/or triangles that are affected by an update operation are identified, there are two methods for altering the current $\mathcal{K}^{(d)}(S)$:

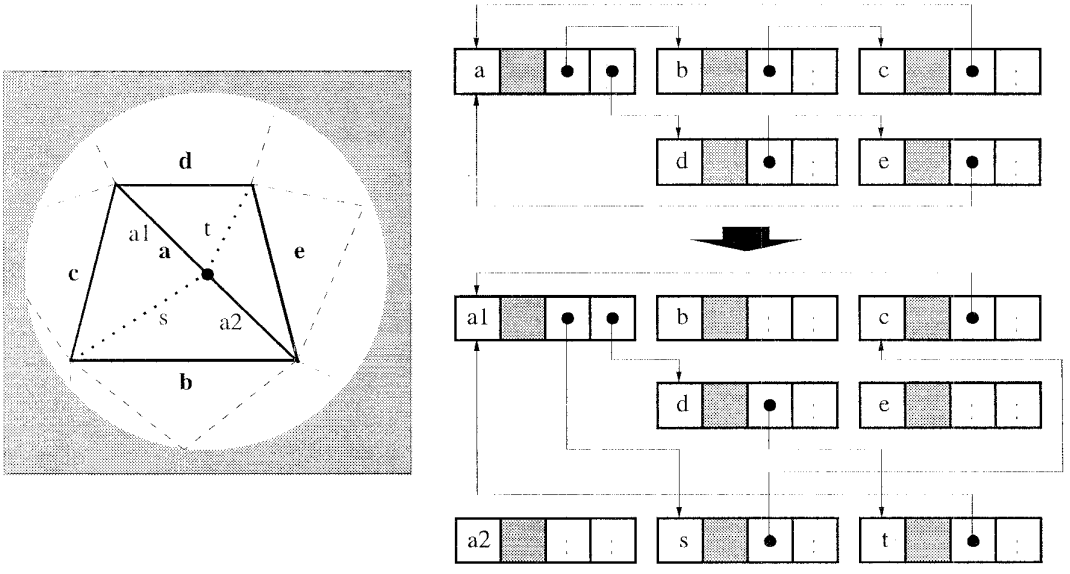

Figure 2. Insertion of a vertex on an edge, and the update of S.

Method A — Keeping all nodes in the current $\mathcal{K}^{(d)}(\mathbf{S})$ unchanged, creating a new lowest level in $\mathcal{K}^{(d)}(\mathbf{S})$, and adding a set new leaf nodes representing appropriate triangles into that level. The new nodes are then linked to the affected triangles.

Method B — Altering the topology of affected triangles in the current $\mathcal{K}^{(d)}(\mathbf{S})$ and deleting and inserting triangles if necessary. The links between these triangles and their parents have to be rebuilt.

Method A is a 'lazy' approach which can be performed quite quickly but may lead to continuing expansion of the size of a $\mathcal{K}^{(d)}(\mathbf{S})$, even when operations such as DELETE_VERTEX are performed. With Method B, the number of levels in a $\mathcal{K}^{(d)}(\mathbf{S})$ remains unchanged. Both require the process of identifying parent triangles and calculating intersections in order to establish new links. Therefore $\mathcal{K}^{(d)}(\mathbf{S})$ must be doubly linked as $\mathcal{K}^{(2)}(\mathbf{S})$. Figure 3 illustrates an example update operation, DELETE_VERTEX, using both methods A and B. With method A, two new triangles ($\mathbf{t}_{15}, \mathbf{t}_{16}$) are added into $\mathcal{K}^{(d)}(S)$, and six links ($\mathbf{t}_{15} \Leftrightarrow \mathbf{t}_2, \mathbf{t}_3; \mathbf{t}_{16} \Leftrightarrow \mathbf{t}_2, \mathbf{t}_3, \mathbf{t}_5, \mathbf{t}_6$) are established. With Method B, two triangles replace four existing ones, and six links ($\mathbf{t}_{15} \Leftrightarrow \mathbf{t}_{10}, \mathbf{t}_{11}; \mathbf{t}_{16} \Leftrightarrow \mathbf{t}_{10}, \mathbf{t}_{11}, \mathbf{t}_{12}, \mathbf{t}_{13}$) are established.

5. Persistent point location

Dynamic point location methods can be made persistent [12, 28, 29] to support a point location query with respect to a subdivision **S** at any time in the past, while updates are allowed in the present version. In other words, a persistent data structure represents all versions simultaneously. It differs from a dynamic structure by maintaining the old version after an update operation (or a set of them). In the following discussion, we focus our interest on partially persistent representations as opposed to fully persistent representations where updates to the past versions are also permitted.

Persistency is usually added to an existing dynamic implementation (which is also called an ephemeral dynamic implementation), ideally with logarithmic access, insertion, and deletion times and O(1) space bounds for insertion and deletion. A persistent implementation of the Kirkpatrick's method that achieves the requirements has been developed by the authors. The discussions on persistent implementations of several other dynamic methods, such as the dynamic slab method and the dynamic separating chain methods, can be found in [5].

There are two types of data structures to be considered for a persistent implementation of the Kirkpatrick's method:

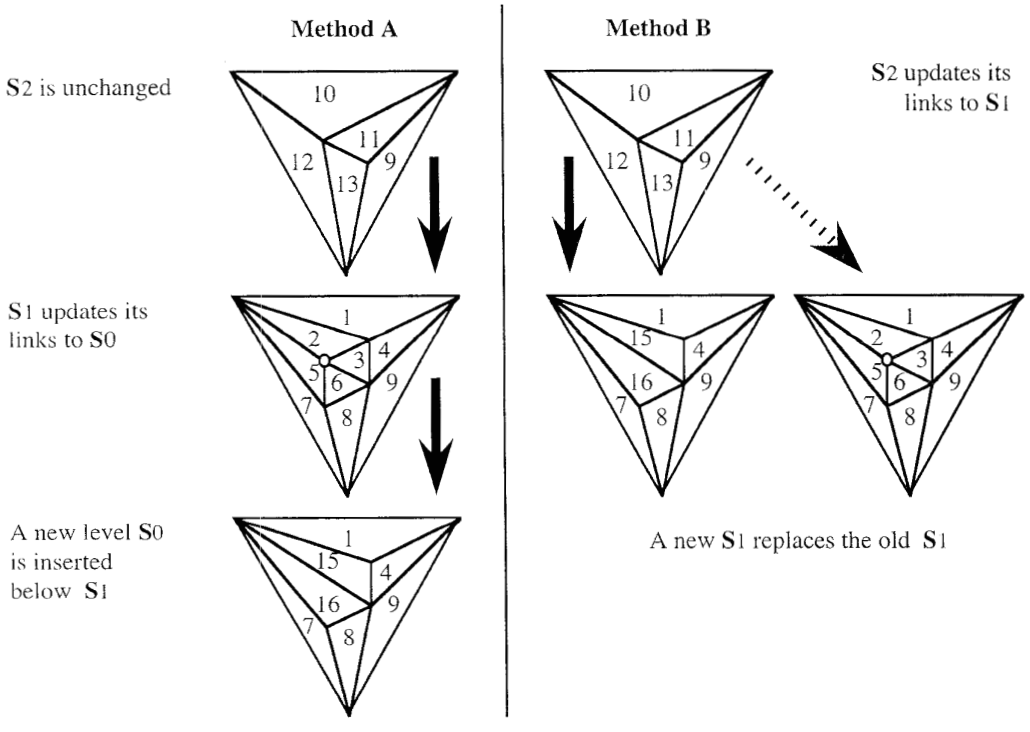

Figure 3. An update operation, DELETE_VERTEX, using methods A and B.

- A $\mathcal{K}$-structure of a persistent subdivision, $\mathcal{K}(\mathbf{S}^{(p)})$ — A special data structure maintains the different versions of a subdivision **S**, but the $\mathcal{K}$-structures built for different versions are not related.
- A persistent $\mathcal{K}$-structure, $\mathcal{K}^{(p)}(\mathbf{S})$ — A special data structure maintains the different versions of a $\mathcal{K}$-structure. In this way, the point location query is persistent but not necessarily the **S**, that is, a query on the past version of **S** can be performed but various versions of **S**, other than the present one, may not be accessed.

In the rest of this section, we are going to concentrate on $\mathcal{K}^{(p)}(\mathbf{S})$, and a detail description of an approach for $\mathcal{K}(\mathbf{S}^{(p)})$ based on Overmars' techniques [29] can be found in [5].

One naive approach would construct a $\mathcal{K}^{(p)}(\mathbf{S})$ by:

1. obtaining **S** from the current $\mathcal{K}^{(p)}(\mathbf{S})$,
2. updating **S**,
3. building a new $\mathcal{K}(\mathbf{S})$,
4. identifying the differences between the new $\mathcal{K}(\mathbf{S})$ and the old $\mathcal{K}^{(p)}(\mathbf{S})$,
5. constructing a new $\mathcal{K}^{(p)}(\mathbf{S})$ with the new $\mathcal{K}(\mathbf{S})$ and the differences.

The different versions of the ephemeral $\mathcal{K}$-structure may differ greatly due to the nature of the Kirkpatrick decomposition, and it would be quite difficult to identify and record the changes from one version to another. This approach is generally inefficient and undesirable, though we have implemented it for the purpose of comparison.

A more appropriate approach is to modify the current $\mathcal{K}^{(p)}(\mathbf{S})$ as the dynamic implementation, and record the modification in a manner such that only certain nodes will be searched for a given version number. The two methods discussed in Section 4.3 derive two different methods for modifying the $\mathcal{K}^{(p)}(\mathbf{S})$, recording the changes and performing version identification. With the issue of persistency, the merits and demerits of the two methods are further differentiated.

Method A — With this method, a new version of $\mathcal{K}^{(p)}(\mathbf{S})$ contains all the nodes in the past versions, and the new nodes are always added into the newly-created lowest level of the $\mathcal{K}^{(p)}(\mathbf{S})$. We simply associate a version stamp (or a time stamp) with each node. In the first version of $\mathcal{K}^{(p)}(\mathbf{S})$, all leaf nodes are stamped as "1", and the rest version "0". New nodes created for the i^{th} version are stamped as "i", as illustrated in Figure 4. Given a version number **k**, the search algorithm will traverses any node with a version stamp $\leq$ **k**. When a stamp greater than **k** is encountered (or inclusion test fails), the search stops descending.

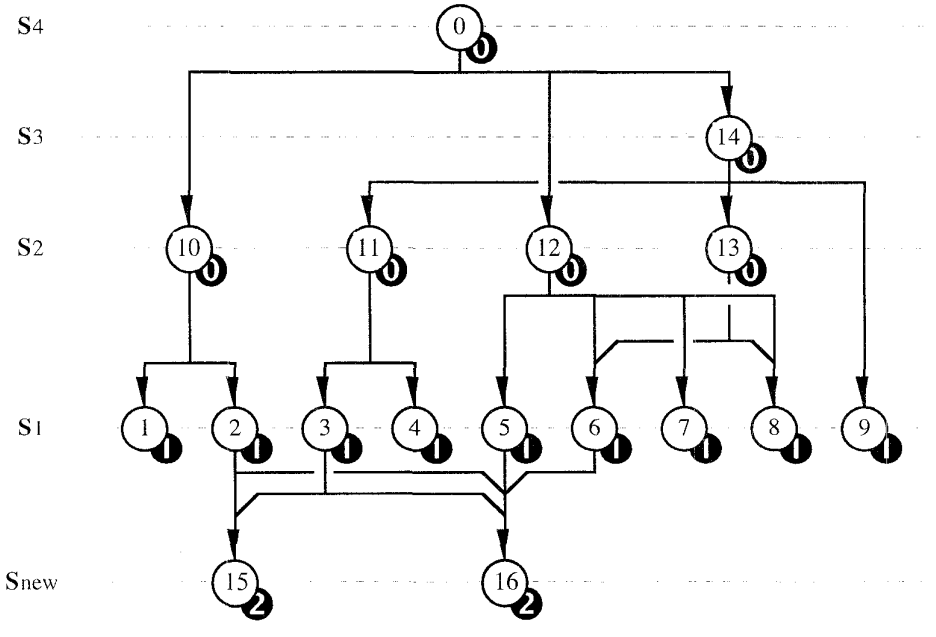

Figure 4. Version stamps in a $\mathcal{K}^{(p)}(\mathbf{S})$ constructed using Method A. The update operation is the same DELETE_VERTEX operation shown in Figure 3.

Method B — The existing Method B for the ephemeral dynamic implementation (described Section 4.3 and Figure 3) can be used for a persistent implementation except that all nodes which are normally to be deleted or replaced in a $\mathcal{K}^{(d)}(\mathbf{S})$ have to be retained in a $\mathcal{K}^{(p)}(\mathbf{S})$. The so-called *fat node* approach which was suggested for partially persistent representations in [28, 29] can be adopted for the modification of a $\mathcal{K}^{(p)}(\mathbf{S})$. Basically, version stamps are added into every node in the $\mathcal{K}^{(p)}(\mathbf{S})$, which is initialised in the same way as in Method A. When an update operation is performed, newly-created nodes are inserted into the current leaf level with a stamp representing the new version number. Old nodes are no longer deleted or replaced. The newly-created nodes are then linked to their parent nodes (which eventually become 'very fat nodes' with many pointers). The depth of a $\mathcal{K}^{(p)}(\mathbf{S})$ remains as a constant but the width of its leaf level expands continually. Figure 5 shows such a $\mathcal{K}^{(p)}(\mathbf{S})$ for the same update operation illustrated in Figure 3.

An efficient and consistent way of handling the pointers for fat nodes is to insert the later version in the front of the list of child nodes. This allows triangles of a later version to be searched first. Triangles which are supposed to have been deleted or replaced will affect the correctness of the search results though they do contribute to the search time. For example, consider the $\mathcal{K}^{(p)}(\mathbf{S})$ shown in Figure 5. Assuming a query point $\mathbf{q}$ in $\mathbf{t}_5$ in the initial version of $\mathcal{K}^{(p)}(\mathbf{S})$, the search algorithm follows a path $\{\mathbf{t}_0\ \mathbf{t}_{12}\ \mathbf{t}_5\}$ to locate $\mathbf{q}$. After the update operation which removes $\mathbf{t}_5$ and three other triangles, $\mathbf{q}$ is located in $\mathbf{t}_{16}$, for which a pointer is added into $\mathbf{t}_{12}$. Because this new pointer (of version 2) is located before the pointer for $\mathbf{t}_5$ (of version 1), when a new query for $\mathbf{q}$ arrives, $\mathbf{t}_{16}$ will be found first. However, if a query for $\mathbf{q}$ is made in conjunction with version number 1, $\mathbf{t}_{16}$ will be ignored as its stamp > 1, and $\mathbf{t}_5$ will be located.

6. Results and remarks

We present the results of the three major extensions to the Kirkpatrick's method in the forms of theorems, analytic comments and run-time data collected from computational experiments. All computational experiments were carried out with an identical set of subdivisions with varying degree of refinement and number of vertices, and due to the limited space, only those most important data-sets are presented. The detail discussions on theorems and their proofs can be found in [5].

6.1 Point location for close successive query point

Theorem 1: *It is possible to have an algorithm for Kirkpatrick's method with O(1) query time in the average case, O(N) space bound and O(N) pre-processing time, provided that successive query points are sufficiently close to each other. The worst case query time is O(log N).*

Although one can easily say that the point location for close successive query point is counter-productive if the distance is too large or less predictable, we can improve this by having an automatic switching strategy depending on the distance between two successive query points or the number of levels of the $\mathcal{K}^{(2)}(\mathbf{S})$ reached by the search

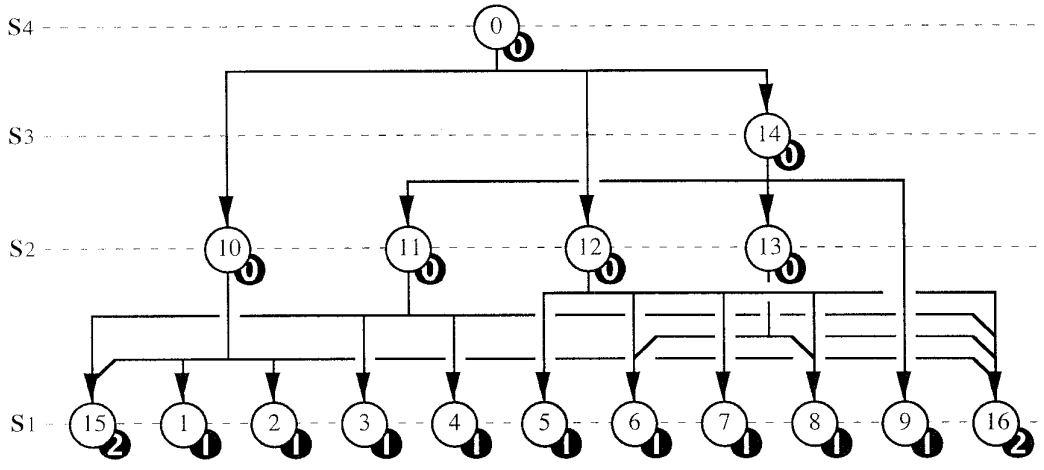

Figure 5. Version stamps in a $\mathcal{K}^{(p)}(S)$ constructed using Method B. The update operation is the same DELETE_VERTEX operation shown in Figure 3.

process. So, an average O(1) query time can be achieved for a group of reasonably close successive query points, but in the worst case O(log N). By having such a switching mechanism, the situation where the total computational cost of using $\mathcal{K}^{(2)}(\mathbf{S})$ exceeds that of the conventional $\mathcal{K}(\mathbf{S})$ can be avoided. The switching can be easily implemented since $\mathcal{K}^{(2)}(\mathbf{S})$ can be used for both the arbitrarily distributed query points and close successive query points.

It would be desirable to have a quantifiable closeness value which could guarantee an O(1) query time for any group of successive query points distributed within a region defined by the value. However, it is obvious that there is no such a value, as one can easily imagine a set of points closely distributed near the edge between triangles t_2 and t_5 in Figure 1. Regardless of the region containing these points, such a sequence would require an O(log N) query time, so long as the points are located in the two triangles in an alternated manner. Tests on many randomly-generated sequences have shown that an average O(1) query time can be achieved when the average distance between points is less than $\mathbf{c}\cdot\sqrt{\mathbf{A}}$, where $\mathbf{A}$ is the average area of triangles in $\mathbf{S}_1$ and $\mathbf{c}$ is a positive number much less than the height of the search tree h(N) [5].

The storage requirement for this extension is also minimal. For each child pointer (pointing to a lower level) in a $\mathcal{K}(\mathbf{S})$, a parent pointer is required by the child node in a $\mathcal{K}^{(2)}(\mathbf{S})$. Therefore, the space bound of the extension remains to be O(N) as in the original method.

6.2 Dynamic point location

Generally, the update operations on a $\mathcal{K}^{(d)}(\mathbf{S})$ involve two main steps:

1. Locating the vertices/edges/triangles that are affected by the update.
2. Modifying the $\mathcal{K}^{(d)}(\mathbf{S})$.

In all cases, the overall performance of the operation depends strongly on Step 1, especially for large subdivisions. The influences of Step 2 become sufficient only with small subdivisions. Step 1 can be carried out in O(log N) time if the operation is specified by using a pick device (such as a mouse), or O(1) if the identifiers of the concerned elements are given.

Theorem 2: *For a $\mathcal{K}^{(d)}(\mathbf{S})$, a single update operation (which can be any operation specified in Section 4.1) requires O(log N) time and O(1) space.*

The main drawback of Method A for the dynamic implementation is that the deleted or replaced triangles remain in the $\mathcal{K}^{(d)}(\mathbf{S})$, and the height of the $\mathcal{K}^{(d)}(\mathbf{S})$ can become quite uncontrollable particularly when highly dynamic and localised updates take place. With Method B, the triangles to be deleted are discarded from the $\mathcal{K}^{(d)}(\mathbf{S})$ and replaced by the new ones. It is more economic with space, but at the cost of speed. This can be easily observed from Table 1 which lists the average update times for different update operations required by simply rebuilding $\mathcal{K}(\mathbf{S})$, using Methods A and B respectively. The programs were written in C, and the tests were carried out on a DEC alpha computer, under a UNIX operating system.

6.3 Persistent point location

Theorem 3: *For a $\mathcal{K}^{(p)}(\mathbf{S})$, a single update operation requires O(log N) time and O(1) space, where N is the number of vertices in the latest version of $\mathbf{S}$. A persistent point location query in a $\mathcal{K}^{(p)}(\mathbf{S})$ takes O(log M) time and requires O(M) space, where M is the total number of vertices in $\mathcal{K}^{(p)}(\mathbf{S})$.*

In the two methods for modifying a $\mathcal{K}^{(p)}(\mathbf{S})$, the same query times of O(log M) and space bounds of O(M) be achieved since the same point location algorithms with a slight modifications and the same organisation of the $\mathcal{K}^{(p)}(\mathbf{S})$ are used. When the number of updates is a small figure in relation to the size of $\mathbf{S}$, M is of O(N). Similar to the dynamic implementation, a single update operation can be carried out in O(log N) time if the operation is specified by using a pick device, or O(1) if the identifiers of the concerned elements are given.

The drawback of Method A is that the height of a $\mathcal{K}^{(p)}(\mathbf{S})$ can become high, and that for Method B is the some nodes in the level $\mathbf{S}_2$ of the $\mathcal{K}^{(p)}(\mathbf{S})$ can be very 'fat', especially when updates are very closely positioned.

6.4 Concluding remarks

To compare our extensions with the existing algorithms, we note that

- Point location for close successive query points is a new way of achieving faster point location query with an O(1) time for reasonably close successive query points.
- The dynamic methods of Cheng and Janardan [12], Goodrich and Tamassia [15], and Baumgarten, Jung and Mehlhorn [17] have the same update times and requirements but slower query times. The method by Chiang and Tamassia [13] achieves the same query time and update time, but requires more space.
- Tamassia [16] has achieved the same query time and space but a slightly slower update time and is limited to triangulation with a restricted set of update operations. Our dynamic method can be generalised for subdivisions other than triangulation with a more flexible set of update operations.
- Goodrich and Tamassia [15] with their semi-dynamic method only achieve better performance in the insertion of a vertex.

Table 1. The average update times of six update operations, which were obtained using three different methods, namely rebuilding the $\mathcal{K}(S)$, modifying the $\mathcal{K}(S)$ using Methods A and B respectively.

Size of S		Method	Average Update Time (μsec) Per Operation					
No. of Vertices	No. of Edges		INSERT VERTEX	MOVE VERTEX	DELETE EDGE *	DELETE EDGE **	INSERT EDGE ***	INSERT EDGE ****
29	36	Rebuild	18035	9142	7964	5002	16105	36697
		Modify A	176	1278	68	2489	1191	820
		Modify B	331	1562	234	3172	1405	1581
62	92	Rebuild	44700	36161	32308	18212	37469	93110
		Modify A	185	1376	107	2508	2168	1269
		Modify B	410	1659	234	3201	2811	2420
256	414	Rebuild	187685	165608	160405	149933	172263	224813
		Modify A	478	1415	127	2372	2479	1679
		Modify B	683	1640	303	3299	3045	2792
525	938	Rebuild	419768	382247	392079	351167	430494	439021
		Modify A	908	1435	166	2508	4509	2069
		Modify B	995	1659	332	3348	4900	3650
1009	1920	Rebuild	763865	792035	793099	798578	850829	925608
		Modify A	1825	1532	351	2743	14308	5075
		Modify B	1933	1757	498	3601	16540	8237
2065	3934	Rebuild	1442134	1714957	2023928	1844918	1849866	2051472
		Modify A	3641	1757	537	3133	29169	7437
		Modify B	3709	2108	761	3777	28994	12230

* keeping endpoints; ** removing endpoints as well; *** removing intersected edges; **** keeping intersected edges and inserting vertices for intersected points.

- The only persistent method by Cheng and Janardan [12] achieves a better update time. However, the method, as described by the authors themselves, is mainly of theoretical interest because it involves rather complex manipulations of data structures.
- We focused our work on practical efficiency as well as theoretical efficiency. We implemented all extensions described in this papers and performed necessary computational tests. In the past, only Edahiro, Kokubo and Asano [4], and Kagami, Edahiro and Asano [30] have reported their implementation and testing for some static methods.
- Most of our methods utilise the same data structures and the same point location algorithm of the original static method. We have: $\mathcal{K}^{(p)}(\mathbf{S}) \supset \mathcal{K}^{(d)}(\mathbf{S}) \supset \mathcal{K}^{(2)}(\mathbf{S}) \supset \mathcal{K}(\mathbf{S})$. But the time and space complexities remain the same, or, in the case of the first extension, is improved.
- These extensions, especially the first two, can be easily implemented without over-complicated data structures. This reiterates the practicality of the methods. The applications of the methods include geographical information systems, scientific visualisation, virtual reality and particle systems [5].

References

[1] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*, Springer-Verlag 1985.

[2] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, Springer-Verlag 1987.

[3] F. P. Preparata. Planar Point Location Revisited, *International Journal of Foundation of Computer Science*, 1(1):71-86, March 1990.

[4] M. Edahiro, I. Kokubo, and T. Asano. New Point-Location Algorithm and Its Practical Efficiency - Comparison With Existing Algorithms, *ACM Transaction On Graphics*, 3(2):86-109, April 1984.

[5] A. Z. B. Haji Talib. *On The Point Location Problems With Applications To Scientific Visualisation*, Ph.D. Thesis, University Of Wales Swansea, 1995.

[6] D. P. Dobkin and R. J. Lipton. Multidimensional Searching Problems, *SIAM Journal on Computing* , 5(2):181-186, 1976.

[7] D. G. Kirkpatrick. Optimal Search in Planar Subdivisions, *SIAM Journal on Computing*, 12(1):28-35, 1983.

[8] H. Edelsbrunner, L. J. Guibas and J. Stolfi. Optimal Point Location in a Monotone Subdivision, *SIAM Journal on Computing*, 15(2):317-340, 1983.

[9] D. T. Lee and F. P. Preparata. Location of a Point in a Planar Subdivision and Its Application, *SIAM Journal on Computing*, 6(3):594-606, Sept. 1977.

[10] F. P. Preparata and R. Tamassia. Dynamic Planar Point location With Optimal Query Time, *Theoretical Computer Science*, 74(1):95-114, 1990.

[11] F. P. Preparata and R. Tamassia. Fully Dynamic Point Location in a Monotone Subdivision, *SIAM Journal on Computing*, 18(4):811-830, Aug. 1989.

[12] S. W. Cheng and R. Janardan. New Results On Dynamic Planar Point Location, *SIAM Journal on Computing*, 21(5):972-999, Oct. 1992.

[13] Y. J. Chiang and R. Tamassia. Dynamisation of the Trapezoid Method For Planar Point Location, *Proceedings of the 7th. ACM Symposium on Computational Geometry*, 1991.

[14] F. P. Preparata. A New Approach To a Planar Point Location, *SIAM Journal on Computing*, 10(3):473-483, Aug. 1981.

[15] M. T. Goodrich and R. Tamassia. Dynamic Trees and Dynamic Point Location, *Proceedings of the 23rd. Annual symposium on Theory of Computing*, 523-533, 1991.

[16] R. Tamassia. An Incremental Reconstruction Method For Dynamic Planar Point Location, *Information Processing Letters*, 37:79-83, 1991.

[17] H. Baumgarten, H. Jung and K. Mehlhorn. Dynamic Point Location in General Subdivisions, *Proceedings Of The Third Annual ACM-SIAM Symposium on Discrete Geometry*, 250-258, 1992.

[18] H. Edlesbrunner, M. H. Overmars and R. Seidel. Some Methods of Computational Geometry Applied to Computer Graphics, *Computer Vision, Graphics and Image Processing*, 28(1):92-108, 1984.

[19] M. H. Overmars. Computational Geometry and Its applications to Computer Graphics, *Advances in Computer Graphics V*, W. Purgathofer and J. Schönhut (Eds.), (75-107) Springer-Verlag, 1989.

[20] D. P. Dobkin. Computational Geometry and Computer Graphics, *Proceedings of The IEEE*, 80(9):1400-1411, Sept. 1992.

[21] H. Samet. Neighbour Finding Techniques For Images Represented By Quadtree, *Computer Graphics & Image Processing*, 18(1):37-57, Jan. 1982.

[22] A. S. Glassner. Maintaining Winged-edge Models, in T. Arvo (Ed.), *Graphics Gem II*, (191-201), Academic Press, 1991.

[23] P. A. Wawrzynek and A. R. Ingraffea. An Edge-Based Data Structure For Two-Dimensional Finite Element Analysis, *Engineering with Computers*, 3:13-22, 1987.

[24] L. Guibas and J. Stolfi. Primitives For The manipulation of General subdivision And The Computation of Voronoi Diagrams, *A C M Transaction On Graphics*, 4(2):74-123, 1985.

[25] O. Palacios-Velez and B. C. Renaud. A Dynamic Hierarchical Subdivision Algorithm for Computing Delaunay Triangulations and Other Closest-Point Problems, *ACM Transaction on Mathematical Software*, 16(3):275-292, Sept. 1990.

[26] Z. Jain-ming, Z. Ke-ran, Z. Ke-ding, and Z. Qiong-hua. Computing Constrained Triangulation and Delaunay Triangulation: A New Algorithm, *IEEE Transactions On Magnetics*, 26(2):694-697, March 1990.

[27] H. Samet, *The Design and Analysis of Spatial Data Structures*, Chapter 4:287-313, Addison Wesley, 1989.

[28] N. Sarnak and R. Tarjan. Planar Point Location Using Persistent Search Tree, *Communication of the ACM*, 29:669-679, 1986.

[29] J. R. Driscoll, N. Sarnak, D. D. Sleator and R. E. Tarjan. Making Data Structures Persistent, Journal Of Computer and System Science, 38:86-124, 1989.

[30] S. Kagami, M. Edahiro and T. Asano. Practical Efficiencies of Point Location Algorithms, *IEICE Transaction on Fundamentals of Electronic Communications and Computer Sciences*, E77A(4):608-614, 1994.