Computing a Threaded Quadtree (with links between neighbors) from the Delaunay Triangulation in Linear Time*

(Extended Abstract)

Drago Krznaric Christos Levcopoulos
Department of Computer Science
Lund University, Box 118, S-221 00 Lund, Sweden.

drago@dna.lu.se christos@dna.lu.se

1 Introduction

A quadtree or a two-dimensional trie is a data structure which can be used for hierarchical representation of a set of points. Hierarchical data structures are of great importance for representing geometric data, with applications in computer graphics, image processing, and robotics [7]. In addition, they can assist important algorithms in computational geometry. It was mainly this latter purpose which motivated us to propose a threaded quadtree. There are several papers considering variants of quadtrees and related data structures, but for a more comprehensive exposition we refer to [7].

Let S be a set of n points (called vertices) in the plane. A quadtree is obtained by first enclosing all vertices with a square. This square is partitioned into four squares, which are each recursively partitioned into four squares, and so on. A recursion ends when a square contains at most one vertex. The induced squares are represented as nodes in a rooted tree, each square having its (at most four) nonempty subsquares as children. However, in the threaded quadtree which we propose, not all squares with only one nonempty subsquare are represented. In this way, the size of the threaded quadtree becomes linear, although it still contains all essential information.

A common operation on a quadtree is, given a node corresponding to a square s, search for all other nodes whose corresponding squares lie in the vicinity of s. To support such operations efficiently we thread (link) nodes that correspond to neighboring squares of the same size. The resulting threaded quadtree has the ability to answer certain types of non-trivial range queries in constant time. This was proved helpful in [4] for an algorithm that computes an approximation of the known complete linkage clustering (in linear time from the Delaunay triangulation). It was also helpful in order to develop an algorithm that, given the Delaunay triangulation, computes the greedy triangulation in linear time [5]. That algorithm also illustrates a reason to start from the Delaunay triangulation, because not only most algorithms for greedy triangulation, but also other algorithms in computational geometry, build the Voronoi diagram (from which the Delaunay triangulation can be obtained trivially).

To describe one type of problems where the threaded quad-tree can be used, suppose that every vertex carries some vital data. In our case, for simplicity, we assume that data is a color. Consider a square-grid covering the plane, and suppose that each square possesses a function of the vertices contained in it. The function might, for example, count the number of vertices of each color (thus determining the color of the square). Imagine now that we start with a fine-grained grid such that every square contains at most one vertex, and then, stepwise in phases, we let the grid become sparser by merging four squares into one until all vertices of S are contained in a single square. (This can be illustrated by repeatedly shrinking a picture represented by p pixels to one that is represented by $p/4$ pixels.) A conceivable task could be to color the nonempty squares at each phase according to their function (the background is assumed to have some default color), and to explore adjacencies. To accomplish that we need at least three pieces of information at every phase: (1) which squares are nonempty, (2) which vertices are contained in a nonempty square, and (3) which nonempty squares are in the vicinity of a nonempty square.

The threaded quadtree is defined in Section 2. In Section 4 we show that, given the Delaunay triangulation of S, the threaded quadtree can be computed in time $O(n\,\alpha(n,n))$, where $\alpha(n,n)$ is the inverse of Ackermann's function. In addition, by taking advantage of certain properties of the Delaunay triangulation, we can show that the running time can be reduced to $O(n)$. Our construction of the threaded quadtree is similar to the clustering process indicated in the above "coloring" example, and the resulting quadtree contains essentially the three pieces of information mentioned at the end (in particular, the third operation can be carried out in constant time due to the threads). For the type of problems illustrated by the example, we describe in Section 5 a general way in which the threaded quadtree can be used.

2 Defining the threaded quadtree

Definition 2.1 *Let d be the distance between the two closest vertices of S. For any integer $i \geq 0$, d_i is defined to be equal to $2^{i-1}d$.*

*This paper was partially supported by TFR

Definition 2.2 *Let H be a square with sides parallel to the coordinate axes that properly contains all vertices of S and that can be partitioned into $2^k \times 2^k$ (k being an integer) equal-sized squares of diameter d_0. Then the d_i-grid is the partitioning of H into $2^{k-i} \times 2^{k-i}$ equal-sized squares of diameter d_i. The squares induced by the d_i-grid are called d_i-squares.*

We say that a d_i-square is *empty* if it contains no vertex; in addition, we assume w.l.o.g. that no vertex is at the boundary of a d_i-square (a vertex lying on a vertical line of a d_i-grid is considered to be contained in the adjacent d_i-square to the right of this line, and, similarly, a vertex lying on a horizontal line is considered to be contained in the adjacent d_i-square above this line; i.e., as if H has been infinitesimally transposed to the left and downwards.)

The easiest way now to define a quadtree would be to let the root correspond to the smallest d_i-square containing all vertices in S, and to define the other nodes recursively so that each node which corresponds to a nonempty d_i-square s has, if $i > 0$, one child for each nonempty d_{i-1}-square contained in s (thus the leaves and the nonempty d_0-squares coincide). This quadtree may however have much more than a linear number of nodes, because it may contain long paths on which each interior node has only one child. A simple refinement that reduces the size to linear is to represent only those d_i-squares for which the corresponding nodes have more than one child, that is to compress each of the above mentioned paths into a single edge. Indeed, we do not loose any essential information by such compressions.

A weakness which remains even in the compressed quadtree is that for two d_i-squares s and q represented in the quadtree, the nearest common ancestor of their corresponding nodes may be the root even if s and q are close to each other. Consequently, to find q given only s, we may have to follow a long path in the quadtree. Therefore, we thread (link) nodes that correspond to d_i-squares that are relatively close to each other.

Definition 2.3 *The* neighborhood *of a d_i-square s consists of all d_i-squares that are within distance less than or equal to d_i from s (the 24 d_i-squares closest to s). The* neighbor list *of a nonempty d_i-square s consists of pointers to all nonempty d_i-squares in the neighborhood of s.*

In the threaded quadtree, for a node corresponding to a d_i-square s, we also have a node for each nonempty d_i-square in the neighborhood of s. Although this means that we now may have more nodes than in the compressed quadtree, we can still linearly bound the number of nodes.

Definition 2.4 *A set of (at most four) nonempty d_i-squares is called a* family *if they have a common point in the plane and no other nonempty d_i-square is within distance less than $4d_i$ from any of them.*

Indeed, the d_i-squares of a family can only have pointers to each other in their neighbor lists. It can be shown (Corollary 4.8) that, for a path compressed to a single edge, all nodes on the path but a constant number correspond to d_i-squares which belong to families.

Definition 2.5 *A nonempty d_i-square s is said to be* significant *if it has at least one of the following properties: (1) s does not belong to a family, or (2) s belongs to a family whose d_i-squares contain together more than four nonempty d_{i-1}-squares.*

The second property in the above definition is merely to ensure that each node has at most four children.

Definition 2.6 *Let m be the smallest integer such that all vertices of S are contained in a single d_m-square. The* threaded quadtree or TQT *of S is a rooted tree, where the root corresponds to the nonempty d_m-square and the leaves correspond to all nonempty d_0-squares. The other nodes correspond to all significant d_i-squares for $i = 1, 2, \ldots, m-1$. Let v be any node of the tree different from the root, and let s be its corresponding d_i-square. Further, let j be the smallest integer such that s is properly contained in a significant d_j-square q. Then the parent of v is the node that corresponds to q. In addition, the node v has also pointers to all nodes that correspond to nonempty d_i-squares in the neighborhood of s.*

3 Computing the leaves

To gain intuition about the algorithm, we can consider how all nonempty d_0-squares are computed. Since d_0 is smaller than the distance between the two closest vertices, there is at most one vertex contained in any d_0-square. Thus we can easily create a list consisting of all nonempty d_0-squares by letting each vertex constitute a nonempty d_0-square.

Let s be any nonempty d_0-square and let v be the vertex in s. The neighbor list of s can be created by a breadth-first search from v on the Delaunay triangulation of S, in such a way that the search ends when all paths of length $< 8d_0$ that emanate from v have been visited. Whenever we visit a vertex contained in a d_0-square which is in the neighborhood of s, we insert a pointer to this d_0-square in the neighbor list of s (unless such a pointer already exists). To see that this suffices for computing the neighbor list of s, let q be any nonempty d_0-square in the neighborhood of s and let u be the vertex in q. Since the diameter of a d_0-square is equal to d_0 we have that $|u, v| \leq 3d_0$. Hence, by the following observation, which was proved in [3], we infer that q is included in the neighbor list of s.

Theorem 3.1 [3] *There is a path of Delaunay edges of total length less than $2.42|u, v|$ between any two vertices u and v.*

Since d_0 is smaller than the distance between any pair of vertices, there is a constant number of vertices within distance less than $8d_0$ from any vertex. Thus we realize

that the neighbor lists of all nonempty d_0-squares can be computed in total time $O(n)$. This subsection is summarized in the following lemma.

Lemma 3.2 *The leaves of the threaded quadtree and their neighbor lists can be computed in time $O(n)$.*

4 Computing the threaded quadtree in a hierarchical way

We use the hierarchical clustering method in [4] to decompose S into subsets for which the TQT can be computed locally. The *rectangular diameter* of a vertex set A, abbreviated rdiam(A), is defined to be the diameter of the smallest enclosing rectangle with sides parallel to the coordinate axes.

Definition 4.1 *A subset A of S is a* 23-cluster *if the distance between vertices of A and vertices of $S - A$ is greater than 23·rdiam(A) or A equals S.*

In [4] it was shown that any two non-identical 23-clusters are either disjoint or one of them is a proper subset of the other. This property causes the 23-clusters to form in a natural way a unique hierarchy, which can be described by the following rooted tree.

Definition 4.2 *The* 23-cluster tree *of S is a rooted tree whose nodes correspond to distinct 23-clusters, where the root corresponds to the vertex set S and the leaves to single vertices of S. Let a be any internal node and let A be its corresponding 23-cluster. Then the children of a correspond to all 23-clusters C such that $C \subset A$ and there is no 23-cluster B such that $C \subset B \subset A$.*

In the continuation we will by a 23-cluster also refer to its corresponding node of the 23-cluster tree and vice versa. An algorithm that computes the 23-cluster tree in $O(n)$ time from a Euclidean minimum spanning tree (or the Delaunay triangulation [1]) of S was given in [4]. As a byproduct of that algorithm we also receive for each 23-cluster A the shortest diagonal which has one extreme in A and the other in a sibling of A.

Definition 4.3 *A 23-cluster A is said to have* grid size i *if i is the smallest integer greater than or equal to 0 such that four d_i-squares with a common point in the plane contain all vertices of A. The (at most four) nonempty d_i-squares that contain vertices of a 23-cluster A of grid size i are called the* covering squares *for A.*

Having computed the leaves, we compute the TQT by a depth-first search from the root of the 23-cluster tree, and for each 23-cluster we compute TQT locally when we backtrack from it. In this way, when we start to compute the TQT of a 23-cluster A, we have already computed the TQT of each child of A. This means that we for each child B have four or less rooted trees, each tree being a subtree of the final TQT and rooted at a node which corresponds to a covering square for B. Hence, in order to compute the TQT of A, it is enough to compute the part of the final TQT which lies between the covering squares for A and the covering squares for the children of A.

The Delaunay triangulation is used to find all nonempty d_i-squares that are within some bounded distance from a nonempty d_i-square s, by searching out on Delaunay edges of a certain length that emanate from s (in a similar fashion as in Section 3).

Definition 4.4 *The* bounded Delaunay edges *of a nonempty d_i-square s, denoted by BD(s), consists of all Delaunay edges of length less than $8d_{i+1}$ such that they have one endpoint in s and the other endpoint outside the neighborhood of s.*

A property of 23-clusters is that they are quite separated from each other. Indeed, the following observation, which is easy to verify, says that they are enough separated so that the bounded Delaunay edges are within a 23-cluster A when computing the TQT of A.

Observation 4.5 *If A is a 23-cluster tree of grid size $i > 0$, then any Delaunay edge connecting a vertex of A with a vertex of $S - A$ has length greater than $8d_i$.*

The following is another property from [4] that we will use.

Lemma 4.6 *For any 23-cluster A with $m \geq 2$ children, let l be the length of the shortest diagonal connecting two children of A, and let l' be the length of the longest diagonal connecting two children of A. Then the ratio between l' and l is less than $25^{m-1}/23$.*

4.1 Computing the threaded quadtree of a 23-cluster

Let A be any 23-cluster and let m be the grid size of A. The TQT of A is computed in a sequence $p_k, p_{k+1}, \ldots, p_{m-1}$ of phases. The objective of a phase p_i is to find all significant d_{i+1}-squares which contain vertices of A. The first phase p_k can be chosen so that no d_k-square is significant, but within a few (constant) number of phases we will start to find significant squares (for example, we can choose k so that $l \in [8d_k, 8d_{k+1})$, where l is the length of the shortest diagonal connecting two children of A).

It is quite easy at a phase p_i to find all significant d_{i+1}-squares if we have all d_i-squares that are relevant for this purpose. The main difficulty is to compute the neighbor list of a nonempty d_{i+1}-squares s, because the neighbor lists of the d_i-squares contained in s do not necessarily contain all nonempty d_i-squares in the neighborhood of s (since the neighborhood of a d_{i+1}-square is larger than of a d_i-square). However, by Theorem 3.1, we know that there is path in the Delaunay triangulation of length less than $8d_{i+1}$ from a vertex in s to a vertex in a d_{i+1}-square

in the neighborhood of s. This is the reason for computing the bounded Delaunay edges in Definition 4.4.

Since we compute bounded Delaunay edges, we need the d_i-squares containing a child B of A at the phase p_i if there is a Delaunay edge of length less than $8d_{i+1}$ incident to exactly one vertex of B. However, at the phases before p_i we do not have to know about B. Therefore, we assign to each phase p_i the portion of the children of A which can be ignored until phase p_i is reached. More precisely, let B be any child of A, and let l be the length of the shortest diagonal that connects B with another child of A. If $l \in [8d_i, 8d_{i+1})$, then B is assigned to phase p_i. By Observation 4.5, a child of grid size i cannot be assigned to a phase before p_i. Hence, if a child B is assigned to some phase p_i, then at most four d_i-squares contain all vertices of B.

At each phase p_i we keep a list L_i consisting of nonempty d_i-squares which our algorithm needs in order to find significant d_{i+1}-squares and to compute their neighbor lists. One part of the d_i-squares in L_i consists of those d_i-squares computed at phase p_{i-1}, and the other part consists of those d_i-squares that contain vertices of a child assigned to phase p_i. The algorithm for locally computing the TQT of a 23-cluster A is as follows, starting with each list L_i being empty:

Algorithm 1

1. Let m be the grid-size of A, and let p_k be the first phase (see above for determining k).

2. Assign each child of A to a phase as described above.

for $i = k, k+1, \ldots, m-1$ **do**

3. Add to L_i each d_i-square that contains vertices of a child that is assigned to phase p_i.

4. For each s in L_i, compute the set $BD(s)$ and associate s with every vertex v in s such that v is an endpoint of an edge in $BD(s)$.

5. For each s in L_i we construct a set $N(s)$ consisting of all nonempty d_i-squares that are within distance $\leq 2d_{i+1}$ from s. This can be done as follows. First all nonempty d_i-squares that are in the neighbor list of s or that can be reached by an edge in $BD(s)$ are visited. Then the same is repeated recursively at each d_i-square q that has been visited such that q is within distance less than $8d_{i+1}$ from s. When the search ends it follows from Theorem 3.1 that we have visited all nonempty d_i-squares within distance $\leq 2d_{i+1}$ from s.

6. For each s in L_i do the following. Let s' be the d_{i+1}-square that contains s. Then s' is associated with each d_i-square in the neighbor list of s that is also contained in s'. Thereafter, the d_i-squares in s' are removed from L_i, and s' is inserted into list L_{i+1}.

7. For each s' in L_{i+1} we compute the neighbor list of s' in the following way. Pick any d_i-square s which is contained in s' and scan the set $N(s)$: whenever a d_i-square contained in a d_{i+1}-square in the neighborhood of s' is encountered, add a pointer to this d_{i+1}-square in the neighbor list of s' unless such a pointer already exists.

8. Traverse the list L_{i+1} and add those d_{i+1}-squares that are significant to the TQT.

endfor
end Algorithm 1

For the sake of analysis, a d_i-square which is contained in the list L_i when Algorithm 1 computes the TQT of a 23-cluster A is said to be an *active square* on A.

Lemma 4.7 *If A is a 23-cluster with $N \geq 2$ children, then the total number of active squares on A is $O(N)$.*

Proof We say that a merging takes place at phase p_{i+1} if two nonempty d_i-squares are contained in a single d_{i+1}-square. Recall that if a child B of A is assigned to a phase p_i, then at most four d_i-squares containing vertices of B are added to the list L_i at step 3. Hence, the total number of d_i-squares added to L_i at step 3 over all $i = k, k+1, \ldots, m-1$ is $O(N)$, and so the total number of mergings over all phases is also $O(N)$. Let B be any child of A, and let p_i be the phase that B is assigned to. Further, let F be the set of at most four nonempty d_i-squares which contain vertices of B and that are added to the list L_i at phase p_i. Finally, let t be the number d_i-squares in F. Since B is assigned to phase p_i, there is a nonempty d_i-square q, $q \notin F$, within distance less than $8d_{i+1}$ from F. This means that after c subsequent phases, for a sufficiently large constant c, one of the following two situations will occur: (1) there is a set F' of nonempty d_{i+c}-squares which all have a common point in the plane, such that q and the d_i-squares in F are contained in distinct d_{i+c}-squares of F', or (2) considering q and the d_i-squares in F, at least two of them are contained in the same d_{i+c}-square (in which case a merging has occurred). In the first situation we have that F' contains at least $t+1$ d_{i+c}-squares. Moreover, if the final phase is not reached yet, then the vertices in F' cannot constitute a 23-cluster by themselves, and so there is a nonempty d_{i+c}-square q' ($q' \notin F'$) within distance less than $23 \cdot 2d_{i+c}$ from F'. Hence, since at most four squares may have a common point in the plane, we realize that if the constant c is chosen sufficiently large, then the following holds: If a nonempty d_i-square s is contained in a d_j-square for more than $4c$ distinct j's, then for some of these j's, a merging has occurred in a d_j-square which is adjacent or equal to the d_j-square containing s. In this way we associate each active square on A with some merging, but to each merging we associate no more than a constant number of active squares on A. Consequently, as there may occur at most $O(N)$ mergings, the total number of active squares on A is $O(N)$. □

Since the 23-cluster tree has $O(n)$ nodes, we obtain

Corollary 4.8 *The total number of active squares during the computation of the threaded quadtree is $O(n)$.*

Lemma 4.9 *Let A be any 23-cluster with $N \geq 2$ children, and let IN(A) be the set of all Delaunay edges which connect children of A. Then Algorithm 1 computes the threaded quadtree of A in time $O(|\text{IN}(A)|)$ plus the time taken to compute the bounded Delaunay edges of each active squares on A.*

Proof Obviously steps 1, 2 and 3 take total time $O(N)$. Since N is less than or equal to $1 + |\text{IN}(A)|$, steps 1,2 and 3 take total time $O(|\text{IN}(A)|)$. Consider now step 4. If we ignore the time taken to compute the $BD(\cdot)$ sets, then step 4 takes totally constant time per Delaunay edge in $BD(s)$ of each active square s on A. But for a sufficiently large constant c and any integer i, $BD(s) \cap BD(s') = \emptyset$ if s is a d_i-square and s' is larger than a d_{i+c}-square. Hence, since the set $BD(s)$ of an active square s on A may by Observation 4.5 only contain Delaunay edges of IN(A), step 4 takes total time $O(|\text{IN}(A)|)$ plus the time taken to compute the $BD(\cdot)$ sets. By using Lemma 4.7 and by observing that there is a constant number of d_i-squares within distance less than $8d_{i+1}$ from any d_i-square, it is straightforward to realize that the steps 5, 6, 7 and 8 also take total time $O(|\text{IN}(A)|)$ plus the time taken to compute the $BD(\cdot)$ sets. □

Since IN(A) and IN(B) are disjoint for non-identical 23-clusters A and B, we obtain

Corollary 4.10 *We can compute the threaded quadtree in time $O(n)$ plus the time taken to compute the bounded Delaunay edges of each active squares.*

4.2 Computing bounded Delaunay edges

Definition 4.11 *Let A be a 23-cluster. Then the set OUT(A) consists of all Delaunay edges with one extreme in A and the other in a sibling of A. The set IN(A) is the union of all sets OUT(B) such that B is a child of A.*

Let (u, v) be any edge of the Delaunay triangulation of S. Further, let A be the nearest common ancestor of nodes u and v in the 23-cluster tree. Finally, let B and C be the children of A lying on the unique paths from A to u and from A to v, respectively. Then (u, v) clearly belongs to IN(A), OUT(B) and OUT(C). Moreover, we can find the nodes A, B and C in constant time by constructing the trees T' and T'' as follows (these extra trees are used to cope with the problem when a node has many children).

Initially the trees T' and T'' are equal to the 23-cluster tree. Then we do the following for each internal node A_1 of the 23-cluster tree. Let $B_1, B_2, \ldots, B_k$ be the children of A_1. Then we add nodes $A_2, A_3, \ldots, A_k$ and edges $(A_1, A_2), (A_2, A_3), \ldots, (A_{k-1}, A_k)$ in both T' and T'' (thus creating a path from A_1 to A_k). Thereafter we remove edges (A_1, B_i) for $i = 1, 2, \ldots, k$ in both T' and T''. Finally, in T' we add edges (A_i, B_i) but in T'' we add edges (A_i, B_{k-i+1}) for $i = 1, 2, \ldots, k$. After the construction, both T' and T'' have $O(n)$ nodes and each internal node has two children.

Consider now the nearest common ancestor of u and v in T' respectively T''. One of them has B as a child and the other has C as a child. The nearest common ancestor can be found in constant time using $O(n)$ preprocessing and $O(n)$ space (see [2]). Thus we can compute the sets IN(A) and OUT(A) for each 23-cluster A in total time $O(n)$.

Let A be any 23-cluster. Further, let k be the greatest integer such that d_k is less than or equal to the length of the shortest Delaunay edge of IN(A). Finally, let m be the smallest integer such that $2d_m$ is greater than the length of the longest Delaunay edge of IN(A). Then, by Lemma 4.6, the difference between m and k is at most $O(|\text{IN}(A)|)$. Hence, we can in time $O(|\text{IN}(A)|)$ create a sequence $b_k, b_{k+1}, \ldots, b_m$ of buckets such that a bucket b_i contains all edges of IN(A) of length in $[d_i, 2d_i)$. Since IN(A) and IN(B) are disjoint for non-identical 23-clusters A and B, we can compute such a bucket sequence of the Delaunay edges in IN(A) for each 23-cluster A in total time $O(n)$.

Let A be any 23-cluster and consider Algorithm 1 of the previous subsection that computes the TQT of A. Let $b_k, b_{k+1}, \ldots, b_m$ be the bucket sequence of IN(A). At any phase p_i of the algorithm, a Delaunay edge in a set $BD(s)$ is contained in one of the buckets $b_i, b_{i+1}, b_{i+2}, b_{i+3}$. Thus we can compute $BD(s)$ of every d_i-square s in the list L_i by simply scanning those buckets, and distribute the edges among the d_i-squares (each edge is considered no more than four times for this purpose). To find out which d_i-squares that contain the endpoints of an edge, we can use the standard union-find algorithm (see [8]). Since there are n nonempty d_0-squares and $O(n)$ Delaunay edges, we perform at most $n-1$ union operations and $O(n)$ find operations. Thus the total time spent to compute the bounded Delaunay edges of all active squares is $O(n\,\alpha(n, n))$, where $\alpha(n, n)$ is the functional inverse of Ackermann's function. Hence, by Corollary 4.10, we obtain the following theorem.

Theorem 4.12 *Let S be any set of n vertices in the plane. Given the Delaunay triangulation of S, we can compute the threaded quadtree of S by the above method in time $O(n\,\alpha(n, n))$ using $O(n)$ space.*

The bounded Delaunay edges of a d_i-square s can also be obtained by walking on Delaunay triangles along the boundary of s. By keeping track of the shortest Delaunay edge that emanate from s in 8 directions, we can avoid considering long Delaunay edges crossing the boundary of s (more details can be found in [4]). In this way, the bounded Delaunay edges can be computed in total time $O(n)$, which gives the following theorem.

Theorem 4.13 *Let S be any set of n vertices in the plane. Given the Delaunay triangulation of S, we can compute the threaded quadtree of S in $O(n)$ time using $O(n)$ space.*

5 On the usage of the threaded quadtree

A context in which the TQT may be used is when we have an algorithm that makes range queries in regions that stepwise increase in size. The information stored at the nodes can then be computed incrementally in a bottom-up fashion as the size of the query regions increases. An example is given in [4], where it is shown how the TQT can be used to find all clusters in the vicinity of a given cluster during the computation of the complete linkage clustering. Another example can be found in [5] where the TQT is used for computing the greedy triangulation, by retrieving all relevant groups of vertices within distance $< cl$ from a given vertex v, where l is the length of the latest produced greedy edge and c is some sufficiently large constant (the number of such groups is constant in the query region). By keeping track of the shortest diagonal between each pair of such groups, it is possible to determine in constant time if a greedy diagonal of about length l is incident to v. Below we describe a more general way in which the TQT can be used for the above two mentioned cases and their alike.

The *c-neighborhood* of a square s, for any integer c, is defined to be the region of all points lying within L_∞ distance not greater than $c \cdot d$ from s, where d is the side-length of s. The *c-neighbors* of a node v in TQT is the set of all other nodes v', such that the square corresponding to v' lies totally in the c-neighborhood of the square corresponding to v, but the square corresponding to the parent of v' does not lie totally in this c-neighborhood. The c-neighbors of v can be found in time $O(c^2)$ (actually faster if the number of c-neighbors of v is small) as follows: First we observe that if s and q are the squares that correspond to v and a c-neighbor v' of v, respectively, then there are two neighboring squares represented in TQT with side-lengths at most c times the side-length of s, such that s is contained in one of them and q is contained in the other. So the node that corresponds to one of those squares is connected to v and the other to v', both by paths having length at most $\lceil \log_2 c \rceil$. In a similar manner we can show that there are four or less nodes such that any c-neighbor of v can be reached from one of them by a (parent to child) path of length at most $\lceil \log_2 c \rceil + 1$, and such that their corresponding squares lie in each other's neighborhoods (so they are linked to each other in the TQT). (Starting at v and going repeatedly to a parent $\lceil \log_2 c \rceil + 1$ times, we come to a node that corresponds to a square of side-length $\geq 2c$ times the side-length of s, and the c-neighborhood of s may be included in at most four squares of such a size.) Thus by starting at v and traversing TQT up (towards the root) to one of these nodes, we can find all c-neighbors by searching the TQT downwards from these nodes, only considering those nodes whose corresponding squares overlap with the c-neighborhood of v. In this way we can once and for all link all nodes with their c-neighbors in total time $O(c^2 n)$, or if the parameter c varies in a from the beginning unknown way, we can compute the c-neighbors dynamically during the course of the algorithm.

To each node of the TQT a record of information is associated (the record being either empty or containing some initial information) and the node is marked not visited. The first step is to mark all leaves visited and to (possibly) put in their records information that is relevant for them (referring to the coloring example mentioned in the introduction, it could be the color of the vertex in the square which the leaf represents). For a node v we define the *input nodes* of v to be all children of v and all c-neighbors of these children. Now, in a bottom up fashion, whenever all input nodes of a node v have been visited, the node v is marked visited. At the same time v is visited, the record that is associated with v receives information, which is a function of the information of v and all information contained in the records associated with the input nodes of v. Thus, if t is the time needed to compute such a function, the total time is bounded by $O(t \cdot n)$.

In the two cases mentioned before (the complete linkage clustering and the greedy triangulation) the two parameters c and t are constant. For such cases the TQT may be particularly useful, since the total time for using it becomes linear.

Acknowledgment We would like to thank Professor Günter Rote for his helpful comments.

References

[1] D. Cheriton and R. E. Tarjan. *Finding minimum spanning trees.* SIAM Journal of Computation 5(4), 1976, 47-56.

[2] D. Harel and R. E. Tarjan. *Fast algorithms for finding nearest common ancestors.* SIAM Journal of Computation 13, 1984, 338-355.

[3] J. M. Keil and C. A. Gutwin. *The Delaunay triangulation closely approximates the complete Euclidean graph.* Proc. of WADS, Lecture notes in Computer Science, 1992, 47-56.

[4] D. Krznaric and C. Levcopoulos. *Computing hierarchies of clusters in linear time from the Delaunay triangulation.* Tech. Rep. LU-CS-TR:94-138, Dep. Comp. Sci., Lund Univ., 1994.

[5] C. Levcopoulos and D. Krznaric. *The greedy triangulation can be computed from the Delaunay in linear time.* Tech. Rep. LU-CS-TR:94-136, Dep. Comp. Sci., Lund Univ., 1994.

[6] F. Preparata and M. Shamos. *Computational geometry: an introduction.* Springer-Verlag, 1985.

[7] H. Samet. *Applications of spatial data structures.* Addison-Wesley, 1989.

[8] R. E. Tarjan. *Efficiency of a good but not linear set union algorithm.* Journal of the ACM 22, 1975, 215-225.