Dynamic Partition Trees

Haijo Schipper and Mark H. Overmars

RUU-CS-89-25
November 1989

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Dynamic Partition Trees

Haijo Schipper and Mark H. Overmars

Technical Report RUU-CS-89-25
November 1989

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
the Netherlands

Dynamic Partition Trees*

Haijo Schipper† Mark H. Overmars‡

November 1989

Abstract

In this paper we study dynamic variants of conjugation trees and related structures that have recently been introduced for performing various types of queries on sets of points and line segments, like half-planar range searching, shooting, intersection queries, etc. For most of these types of queries dynamic structures are obtained with an amortized update time of $O(\log^2 n)$ (or less) with only minor increases in the query times. As an application of the method we obtain an output-sensitive method for hidden surface removal in a set of n triangles that runs in time $O(n \cdot k^{\log_2(1+\sqrt{5})-1} \log n)$ where k is the size of the visibility map obtained.

1 Introduction

In recent years a class of new data structures, called *partition trees*, has been designed for storing sets of points to answer particular types of queries, in particular half-planar range queries in which one wants to determine which points (or their number) lie on one side of a given line (or inside a given triangle). Partition trees recursively divide the plane into smaller parts in such a way that any line intersects a relative small number of parts and that every part contains a progressively decreasing number of points. In this way one can efficiently detect large parts of the plane which lie totally inside or totally outside the query range.

The first type of partition tree was introduced by Willard [21]. His *near-ideal J-way polygon trees* have a query time for half-planar range searching of $O(n^{\log_6 4}) = O(n^{.774})$ and a worst-case preprocessing time of $O(n^2)$ (although the expected preprocessing time is $O(n \log^2 n)$). Edelsbrunner and Welzl [7, 8] improved

*Research of the second author was partially supported by the ESPRIT II Basic Research Actions Program of the EC, under contract No. 3075 (project ALCOM).

†Department of Computer Science, University of Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands.

‡Department of Computer Science, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands.

this by introducing their *conjugation trees*. This tree has $O(n \log n)$ preprocessing time and a query time of $O(n^{\log_2(1+\sqrt{5})-1}) = O(n^{.695})$. Both structures use linear storage.

Later Haussler and Welzl [9] presented *partition trees based on ϵ-nets*. The query time for this type of tree is $O(n^{\frac{2}{3}+\gamma})$, for any $\gamma > 0$. Recently Matoušek [10] showed how to construct these trees in $O(n \log n)$ time. (Before only a randomized construction algorithm was known.) The most recent and best results are the *optimal spanning trees, with low stabbing number*, presented by Welzl [20]. As shown by Agarwal [1] these can be constructed in time $O(n\sqrt{n}\log^{1+\omega} n)$, for some $\omega < 3.3$. The query time is $O(\sqrt{n})$ (which is optimal due to the lower bound results of Chazelle [3]).

All the structures mentioned above are static structures. However it is sometimes desirable to have a structure which allows for insertions and deletions in a reasonable efficient way. In this paper we will study techniques for obtaining dynamic partition trees. We will restrict ourselves to the dynamic version of the conjugation trees. This is done because the conjugation tree can be constructed efficiently, which is a necessity to get efficient update times using the presented techniques, and conjugation trees are simple to adapt and use for many other types of queries. We will show how to adapt the conjugation tree to a so-called *δ-conjugation tree* such that the partial rebuilding technique as introduced by Overmars [12] can be used to dynamize them. We also investigate how to apply dynamization techniques for decomposable searching problems (as introduced by Bentley [2] and Overmars [12]) to conjugation trees.

In a recent paper [14] it is shown how to store line segments in conjugation trees to perform different types of queries, like e.g. intersection queries and shooting queries. This is accomplished by associating extra data structures to nodes of the conjugation tree, storing subsets of the line segments. We will show that the techniques for dynamizing conjugation trees also apply to these augmented structures. As one of the applications we obtain an output-sensitive solution to hidden surface removal in a set of triangles.

The paper is organized as follows: In section 2 the δ-conjugation tree is introduced. It is shown that queries on such a tree can be performed in almost the same time bounds as for the original conjugation tree. The advantage of δ-conjugation trees is that they have more flexibility. In section 3 we use this extra flexibility to perform updates in the structure using the technique of partial rebuilding. The resulting structure still uses linear storage, has an amortized update time of $O(\log^2 n)$ and a query time of $O(n^{\log_2(1+\sqrt{5})-1+\gamma})$ for arbitrary small $\gamma > 0$.

Next, in section 4 we shown how the techniques for dynamizing decomposable searching problems can be used with conjugation trees. To this end we show how "weak deletions" (see [12]) can be performed on conjugation trees. The resulting structure has an insertion time of $O(\log^2 n)$ and a deletion time of $O(\log n)$ (worst-case). The query time remains of the same order of magnitude. Of course, this

technique can only be applied if the searching problem we want to solve with the structure is decomposable.

As stated above conjugation trees can also be used to answer queries on line segments. To this end two augmented versions of conjugation trees are introduced in [14]: segment conjugation trees and interval conjugation trees. Sections 5 and 6 show how to obtain dynamic equivalents of these structures, using the same techniques.

One of the applications of these dynamic structures is an output-sensitive algorithm for hidden surface removal for a set of triangles in space which we present in Section 7. Although the bound obtained is not very good, the method is interesting and might have other applications as well. Finally, in Section 8 we give some concluding remarks and state some open problems.

2 The δ-conjugation tree

Conjugation trees are based on a partition of the set of points by a line in two equal halves. To introduce more freedom in our structure we will define a δ-conjugation tree in which lines split the set of points in about equal halves. Such a splitting line is called a δ-bisector.

Definition 1 *Let $\mathcal{S}$ be a set of n points in the plane. Let δ be a constant with $\frac{1}{2} \leq \delta < 1$. A line l is called a* δ-bisector *of $\mathcal{S}$ if and only if the number of points to the left of l is at most δn and the number of points to the right of l is also at most δn.*

We also say that *l δ-bisects $\mathcal{S}$*. Obviously, such a δ-bisector always exists. (From now on we assume that $\frac{1}{2} \leq \delta < 1$ without explicitly stating so.) A normal bisector is a $\frac{1}{2}$-bisector.

l splits $\mathcal{S}$ into two subsets. $\mathit{Left}(\mathcal{S}, l)$ is the set of points to the left of l and $\mathit{Right}(\mathcal{S}, l)$ is the set of points to the right of l (if l is horizontal replace left by above and right by below or vice-versa). When l contains points of $\mathcal{S}$ we denote them by $\mathit{On}(\mathcal{S}, l)$. The points in $\mathit{On}(\mathcal{S}, l)$ we distribute among $\mathit{Left}(\mathcal{S}, l)$ and $\mathit{Right}(\mathcal{S}, l)$ in the following way. Let n_l, n_r and n_o denote the number of points to the left, to the right and on l. If $n_l + n_o \leq \delta n$ then we put all point on l in $\mathit{Left}(\mathcal{S}, l)$. Otherwise we put in $\mathit{Left}(\mathcal{S}, l)$ the top $\delta n - n_l$ points on l and in $\mathit{Right}(\mathcal{S}, l)$ the remaining points. Let s_l denote a position on l than separates the points that go to $\mathit{Left}(\mathcal{S}, l)$ from those that go to $\mathit{Right}(\mathcal{S}, l)$. In this way $\mathcal{S}$ is indeed partitioned into two subsets, each with size at most δn.

Definition 2 *Given a set $\mathcal{S}$ of points in the plane and a line l, another line m is called a* δ-conjugate *of l for $\mathcal{S}$ if m is a δ-bisector of both $\mathit{Right}(\mathcal{S}, l)$ and $\mathit{Left}(\mathcal{S}, l)$.*

A normal conjugate is a $\frac{1}{2}$-conjugate. A useful simple lemma on δ-conjugates is the following:

Lemma 2.1 *Given a λ-conjugate m of l for $\mathcal{S}$, then for every $\delta \geq \lambda$, m is also a δ-conjugate of l for $\mathcal{S}$.*

In particular, a $\frac{1}{2}$-conjugate is a δ-conjugate for any δ. Willard [21] has shown that a $\frac{1}{2}$-conjugate always exists, and Megiddo [11] gives an algorithm for finding such a $\frac{1}{2}$-conjugate in $O(n)$ time. Hence, we obtain the following result:

Lemma 2.2 *Given a set $\mathcal{S}$ of n points and a line l, for each $\frac{1}{2} \leq \delta < 1$ a δ-conjugate always exists and can be found in $O(n)$ time.*

Using the concept of conjugates we can now define our data structure which will be called a *δ-conjugation tree.*

Definition 3 *Let $\mathcal{S}$ be a set of n_v points in the plane. Let l_v be a δ-bisector for $\mathcal{S}$. A* δ-conjugation tree *$\mathcal{T}$ for $\mathcal{S}$ and l_v is a binary tree, defined as follows: If $n_v = 0$ then $\mathcal{T}$ is an empty leaf. If $n_v = 1$ then $\mathcal{T}$ is a leaf containing the point. Else, let l' be a δ-conjugate of l_v for $\mathcal{S}$. T consists of:*

1. *A root v storing l_v, s_l, n_v, $On(\mathcal{S}, l_v)$ and n_o (the number of points in $On(\mathcal{S}, l_v)$).*
2. *A left subtree that is a δ-conjugation tree for $Left(\mathcal{S}, l_v)$ and l'.*
3. *A right subtree that is a δ-conjugation tree for $Right(\mathcal{S}, l_v)$ and l'.*

Note that if $\delta = \frac{1}{2}$ the structure described above is the normal conjugation tree given by Edelsbrunner and Welzl [7, 8]. (In Edelsbrunner [6, pp. 345 - 348] the structure is called a ham-sandwich tree.) If $\delta > \frac{1}{2}$ the tree need not be perfectly balanced. This gives us the freedom to perform updates on it. The depth of a δ-conjugation tree however is still bounded by $O(\log n)$ as is stated the following lemma (the trivial proof of which has been omitted).

Lemma 2.3 *The depth of a δ-conjugation tree is bounded by $\lceil \log_{\frac{1}{\delta}} n \rceil = O(\log n)$.*

With each node v of a δ-conjugation tree $\mathcal{T}$ we can associate a part of the plane, called its *cell* or *range*, denoted as ran_v. If v is the root of $\mathcal{T}$ ran_v is the whole plane, else ran_v is the intersection between $ran_{father(v)}$ and the open half-plane to left or right of $l_{father(v)}$. The *boundary* of v, denoted as $bound_v$ is the closure of ran_v.

The following observations immediately follow from the definition of δ-conjugation trees.

Observation 1 *Let v be a node of a δ-conjugation tree. The following properties hold:*

- *All points stored in a subtree rooted at v lie in ran_v or on $bound_v$.*
- *ran_v is a convex area.*

- *$bound_v$ is a, not necessarily closed, connected convex polygon.*

- *$bound_v$ consists of (parts of) bisectors at ancestors of v.*

- *No bisector at an ancestor of v properly intersects ran_v or $bound_v$.*

Note that points can be stored more than once in a δ-conjugation tree. For some nodes v they can be part of both $On(\mathcal{S}, l_v)$ and $Left(\mathcal{S}, l_v)$ (or $Right(\mathcal{S}, l_v)$). This is necessary to keep the tree balanced when updates occur. Still the tree uses only linear storage and can be constructed efficiently:

Theorem 2.4 *Let $\frac{1}{2} \leq \delta < 1$. Let $\mathcal{S}$ be a set of n points in the plane. A δ-conjugation tree $\mathcal{T}$ for $\mathcal{S}$ can be constructed in $O(n \log n)$ time and uses $O(n)$ storage.*

Proof. Edelsbrunner and Welzl [8] have shown how to construct a $\frac{1}{2}$-conjugation tree $\mathcal{T}$ in $O(n \log n)$ time. Using lemma 2.1 $\mathcal{T}$ is also a δ-conjugation tree.

It is obvious that $\mathcal{T}$ has $O(n)$ nodes. As noted above, some points are duplicated and stored more than once. It is even possible that some points are stored up to $\Omega(\log n)$ times. Note however that a point that is stored both in the subtree rooted at v and at an ancestor of v must lie on $bound_v$. Also note that a point that is stored both at v and in the subtree rooted at v must lie on l_v. But l_v can intersect $bound_v$ at most twice, since $bound_v$ is connected and convex (see the above observation). Thus at v there are stored at most two points which are also stored above v and below v in the tree. The storage bound follows. □

The important property of normal conjugation trees is that any line intersects the ranges ran_v for only a small number of nodes v. This is based on the trivial observation that, whenever a line l intersects ran_v, for at least one of the four grandsons of v l does not intersect the range. The same property holds for δ-conjugation trees. We obtain the following result:

Theorem 2.5 *For every $\gamma > 0$ a δ exists such that any line crosses no more than $O(n^{\log_2(1+\sqrt{5})-1+\gamma})$ ranges of a δ-conjugation tree containing n points.*

Proof. Let $C(n)$ be the maximum number of nodes which a line l can cross in a δ-conjugation tree containing n points. Then we have

$$C(n) \leq C(\delta n) + C(\delta(1-\delta)n) + O(1).$$

If $\delta = \frac{1}{2}$ the solution to this recurrence is (see [7, 8]) $C(n) = O(n^{\log_2(1+\sqrt{5})-1})$. It seems hard to give an exact solution to the recurrence (in both n and δ) but for our purposes an estimation is good enough. Let α be such that $C(n) = O(n^{\alpha+\epsilon})$ for every $\epsilon > 0$. Such an α clearly exist. It easily follows that any α such that

$$\delta^\alpha + (\delta(1-\delta))^\alpha \leq 1$$

will do.

Define a function $\alpha(\delta)$, such that $\delta^{\alpha(\delta)}+(\delta(1-\delta))^{\alpha(\delta)}=1$. Because for $\frac{1}{2} \leq \delta < 1$, $\alpha(\delta)$ is continuous and monotone increasing, and $\alpha(\frac{1}{2}) = \log_2(1+\sqrt{5})-1$, it is clear that for any $\gamma' > 0$ there exists a δ such that $\alpha(\delta) < \log_2(1+\sqrt{5})-1+\gamma'$. Now choose $\gamma' = \gamma - \epsilon$ and the lemma follows. □

One of the basic types of queries for which conjugation trees were designed are half-planar range queries: Given a set of points $\mathcal{S}$ in the plane and a line h, report those points (or their number) that lie to the left (or to the right) of h. We can store the points in a δ-conjugation tree $\mathcal{T}$ and answer the query in the following way (see [8]): We search with h in $\mathcal{T}$. At a node v we have three possible cases. If h does not intersect ran_v and ran_v lies to the left of h we report all points in the subtree rooted at v (or we add n_v to a global counter). If h does not intersect ran_v and ran_v lies to the right of h then nothing needs to be done. Otherwise (h intersects ran_v) we continue the search at both sons of v. (Some care has to be taken when h lies on l_v. In this case we use the set $On(\mathcal{S}, l_v)$ to find the correct answers. See [8] for details.)

Theorem 2.6 *Let $\mathcal{S}$ be a set of n points. For each $\gamma > 0$ there exists a δ such that in a δ-conjugation tree for $\mathcal{S}$ half planar range queries can be performed in $O(n^{\log_2(1+\sqrt{5})-1+\gamma}+k)$ time using linear storage, where k is the number of reported answers. Counting queries can be performed in time $O(n^{\log_2(1+\sqrt{5})-1+\gamma})$.*

Proof. Whenever we visit a node v, v is either the root or the range of its father was intersected by h. From theorem 2.5 it follows that we visit at most $O(n^{\log_2(1+\sqrt{5})-1+\gamma})$ nodes. □

3 Performing updates using partial rebuilding

We will now show how the δ-conjugation tree can be updated when insertions or deletions occur. The method applies the technique of *partial rebuilding* introduced by Overmars [12, Chapter 4].

Assume $\mathcal{T}$ is a δ-conjugation tree. We perform updates on $\mathcal{T}$ by simply adding or deleting the point p without rebalancing the structure. Such updates can easily be performed on δ-conjugation trees. We search with p in the tree and add it to (or delete it from) the appropriate $On(\mathcal{S}, l_v)$ sets and insert (or delete) a leaf for the point. To be precise, an insertion proceeds as follows:

Algorithm: Insert

Suppose a point p has to be inserted in a tree $\mathcal{T}$. (p not already present in $\mathcal{T}$.) Let v be the current node, with subtree $\mathcal{T}_v$. Then there are three cases.

1. $\mathcal{T}_v$ is empty. Then a new tree is created, containing p only.

2. $\mathcal{T}_v$ is not empty, and p lies on l_v. Then if p lies above (or below) s_l, p is inserted in $On(\mathcal{S}, l_v)$ and in left subtree of v (or the right subtree of v). Also the values n_v and n_o are adjusted.

3. $\mathcal{T}_v$ is not empty, and p does not lie on l_v. Then if p lies right (left) of l_v, p is inserted in the right (left) subtree of v. Also n_v is adjusted.

The deletion method works in a similar way. Such an operation clearly takes time the depth of the tree, i.e., $O(\log n)$.

Obviously, after such an update the tree might no longer be balanced, i.e., for some nodes v (which must lie on the search path of p) either $Left(\mathcal{S}, l_v)$ or $Right(\mathcal{S}, l_v)$ might have become too large ($> \delta n_v$). We rebalance the tree by rebuilding part of it. The naïve approach would be to rebuild only the subtree below the highest node that became out of balance. Looking at the definition of the δ-conjugation tree it can be concluded that this approach is not correct. When a subtree $\mathcal{T}_v$ rooted at a node v is rebuilt its new bisector l_v^{new} will normally be different from the old bisector l_v^{old}. But because the brother of v must have the same bisector as v, the structure does no longer satisfy the definition of a δ-conjugation tree. Therefore, (if v is not the root) we rebuild both subtrees $\mathcal{T}_v$ and $\mathcal{T}_w$ for the brother w of v. To avoid that we have to rebuild them again soon we rebuild them as $\frac{1}{2}$-conjugation trees.

Such a rebuilding takes time $O(n_v \log n_v)$ because the subtree rooted at w contains about the same number of points as the subtree rooted at v. This can sometimes be very large, for example when v is the root of the tree. Partial rebuilding though is based on the fact that, once a subtree is rebuilt it takes many updates before it has to be rebuilt again (as we will show below). Hence, the amortized update time will be small.

Lemma 3.1 *Let $\delta > \frac{1}{2}$. Let $\mathcal{T}$ be a newly built $\frac{1}{2}$-conjugation tree of n points. It takes $\Omega(n)$ updates on $\mathcal{T}$ before the root v will be out of balance, i.e., before one of the subtrees of v contains $> \delta m$ points, where m is the current number of points in the set.*

Proof. In the worst case insertions only take place in one son of v and deletions in the other (in this way, the root goes out of balance as quickly as possible). Before any updating, both sons contain at most $\lceil \frac{n}{2} \rceil$ points. Suppose v gets out of balance after i insertions and d deletions. Define $u = i + d$ to be the number of updates. Then, because v is not longer in balance

$$\left\lceil \frac{n}{2} \right\rceil + i > \delta(n + i - d)$$

Thus

$$\frac{n}{2} > \delta(n+i-d) - i - 1 \Rightarrow (2-2\delta)i + 2\delta d > (2\delta-1)n - 1$$

As $2\delta > 2 - 2\delta$ (because $\delta > \frac{1}{2}$) we obtain

$$2\delta u > (2\delta-1)n - 1 \Rightarrow u > \frac{2\delta-1}{2\delta}n - 1$$

which is $\Omega(n)$ because $\delta > \frac{1}{2}$. □

So whenever we rebuild the subtree of a node v (together with the subtree at the brother w of v) it take $\Omega(v_n)$ updates in $\mathcal{T}_v$ before we again have to rebuild the subtree a v. Stated differently, whenever we rebuild a subtree at v there have been $\Omega(n_v)$ updates in $\mathcal{T}_v$ at which no work needed to be done on rebuilding at node v. Hence, we can charge the $O(n_v \log n_v)$ work for rebuilding to these updates, making for $O(\log n_v) = O(\log n)$ per update.

The same update will be charged at most once from node v but it might get charged from other nodes as well. All these nodes must lie on the search path of the update. Hence, there can be at most $O(\log n)$ nodes that can charge costs to the update. So each update is charged at most $O(\log n)$ time $O(\log n)$ work. (See [12] for a more extensive description of this technique of charging costs.) We conclude:

Theorem 3.2 *For each $\delta > \frac{1}{2}$ the amortized update time in a δ-conjugation tree is bounded by $O(\log^2 n)$.*

As a result we obtain e.g. the following theorem:

Theorem 3.3 *Let $\mathcal{S}$ be a set of n points. For each $\gamma > 0$ there exists a data structure for storing $\mathcal{S}$ which uses linear storage and can be built in time $O(n \log n)$ such that half planar range queries can be performed in $O(n^{\log_2(1+\sqrt{5})-1+\gamma} + k)$ time, where k is the number of reported answers. Counting queries can be performed in time $O(n^{\log_2(1+\sqrt{5})-1+\gamma})$. Insertions and deletions can be performed in $O(\log^2 n)$ amortized time.*

4 Using decomposability

We will now apply a different dynamization technique that provides even better bounds. The draw-back of the method is that it can only be applied for a certain subclass of searching problems.

Bentley [2] was the first to define the class of *decomposable searching problems* (see also Overmars [12]). Let $PR(x, V)$ denote the answer to the searching problem PR for a query object x and a set V.

Definition 4 *[2] A searching problem PR is called* decomposable *if and only if for any partition $A \cup B$ of V and any query object x,*

$$PR(x, V) = \square(PR(x, A), PR(x, B))$$

for some constant time computable operator $\square$.

For decomposable searching problems it is not necessary to keep all points in one large structure. They can be divided over a number of smaller structures and the answers can be composed using the operator $\square$. Many searching problems are decomposable. For example range searching is decomposable, using the operator $\square = \cup$, which can be performed in constant time because the sets are disjoint.

Many general techniques are known for dynamizing structures for decomposable searching problems (see [12]). Unfortunately, these techniques only yield good insertion time bounds. To be able to perform deletions efficiently as well, Overmars [12] introduces the concept of *weak deletions.*

Definition 5 *[12] Deletions are called* weak *if and only if there exists a constant α, with $0 < \alpha < 1$ such that the routines to carry them out on a newly built (i.e. perfectly balanced) structure of n points guarantee that after $\leq \alpha n$ deletions, the query time, the amount of storage required and the weak deletion time on the resulting structure of m points are no more than a factor k_α worse than of a new built structure of m points, for some constant k_α depending solely on α and the type of structure.*

Weak deletion can easily be carried out on conjugation trees:

Lemma 4.1 *Deletions in a conjugation tree can be performed weakly in time $O(\log n)$.*

Proof. We perform the weak deletion by simply removing the point from the conjugation tree without any rebalancing. Such a deletion takes time the depth of the tree, which is $O(\log n)$ when the conjugation tree is balanced. Take $\alpha = \frac{1}{2}$. After $\leq \alpha n$ deletions we have $m \geq \frac{1}{2}n$. Each line will still cross at most $O(n^{\log_2(1+\sqrt{5})-1})$ ranges in the tree. This should have been $O(m^{\log_2(1+\sqrt{5})-1})$. Because $\log_2(1+\sqrt{5}) - 1 < 1$ the structure is less than twice as bad. Hence, for most types of queries, the query time becomes less than a factor 2 larger. The amount of storage required will become $O(m)$ as required and the weak deletion time remains $O(\log n)$ which is at most one larger than $O(\log m)$. Hence, the deletions are indeed weak. □

Overmars [12] shows that, when a structure allows for weak deletions, it can be turned into a structure that allows for real deletions in almost the same time bounds. The technique is called *global rebuilding* and works by rebuilding the complete structure after αn deletions. For the conjugation tree this results in a deletion time of

$O(\log n)$. (See [12] for a description of the technique.) The resulting structure has still $O(n \log n)$ preprocessing time and requires $O(n)$ storage.

We can now use the decomposability of the query problem PR to obtain a fully dynamic data structure. Let $Q(n)$ be the query time for PR on a conjugation tree. Assuming that $Q(n) = \Omega(n^\epsilon)$ for some $\epsilon > 0$ as is the case for all types of queries that use conjugation trees) we can apply Theorem 7.3.3.2 of [12] to obtain

Theorem 4.2 *Let S denote a set of n points in the plane. Then there exists a data structure for S, which requires $O(n)$ storage and $O(n \log n)$ time for construction, such that PR can be solved in time $O(Q(n))$, insertions take time $O(\log^2 n)$ and deletions $O(\log n)$.*

Note that these are worst-case time bounds. Using the results in [12] also other trade-offs between query and insertion time can be obtained. Applying the above theorem to half planar range searching we obtain the following result:

Theorem 4.3 *Let S be a set of n points. There exists a data structure for storing S which uses linear storage and can be built in time $O(n \log n)$ such that half planar range queries can be performed in $O(n^{\log_2(1+\sqrt{5})-1} + k)$ time, where k is the number of reported answers. Counting queries can be performed in time $O(n^{\log_2(1+\sqrt{5})-1})$. Insertions can be performed in $O(\log^2 n)$ time and deletions in $O(\log n)$ time.*

5 Segment partition trees

Partition trees were introduced to store sets of points and answer particular queries. Recently, it was shown by Overmars, Schipper and Sharir [14] that one can adapt partition trees, by associating particular structures to the internal nodes, such that they can be used to store line segments and answer e.g. intersection queries. In this section and the next we will show that these augmented structures can (often) be dynamized as well.

The first type of structure introduced by [14] is the *segment partition tree*. Let us briefly describe the structure (for details see [14]). Let S be a set of line segments. The segment partition tree has a conjugation tree T on the set of endpoints of line segments in S as underlying structure. At each internal node v a data structure B_v is associated, depending on the query which is to be performed, storing a subset of the line segments. Moreover we store at each node an ordinary segment tree S_v (see e.g. [16] for a description of the segment tree). Line segments are stored in the following way: For each segment s we search in the tree. At each node v one of the following cases can occur:

1. s intersects the cell ran_v and has no endpoints in the interior of ran_v. Then the part of s which intersects ran_v is stored in the associated structure B_v.

2. s lies on the splitting line l_v. In this case s is stored in the segment tree S_v.

3. Otherwise, the search is continued at those sons of v where s (partially) intersects the range.

It is easy to see that each segment will be stored in at most $O(\log n)$ B-structures and in at most 2 S-structures. Now assume the data structure B has a preprocessing time of $P_B(n)$, an (amortized) insertion time of $I_B(n)$, an (amortized) deletion time of $D_B(n)$, a query time of $Q_B(n)$ and it uses $M_B(n)$ storage. The following lemma follows easily (see [14]):

Lemma 5.1 *A segment partition tree $\mathcal{T}$ uses $O(M_B(n)\log n)$ storage and can be constructed in time $O(P_B(n)\log n)$.*

To dynamize segment partition trees we can adapt both of the techniques described in sections 3 and 4. Let us first look at the technique using partial rebuilding.

We replace the underlying conjugation tree $\mathcal{T}$ by a δ-conjugation tree. To perform an insertion of a segment s we first insert the two endpoints of s in $\mathcal{T}$ in the way described in section 3. This might cause part of the tree to be rebuilt as a perfectly balanced segment partition tree. Next we search with s in the tree in the way described above and insert s into the at most $O(\log n)$ B-structures and the at most 2 S-structures. This takes total time $O(I_B(n)\log n)$. (Segment trees allow for insertions and deletions, using partial rebuilding, in $O(\log n)$ amortized time. These bounds can also be obtained as worst-case time bounds.)

If a line segment s has to be deleted we first delete it from all the B- and S-structures it appears in. Next we remove the two endpoints from $\mathcal{T}$, possibly rebuilding some subtrees.

Lemma 5.2 *Insertions in a segment partition tree can be performed in amortized time $O(I_B(n)\log n+\frac{P_B(n)}{n}\log^2 n)$. Deletions take $O(D_B(n)\log n+\frac{P_B(n)}{n}\log^2 n)$ amortized time.*

Proof. Following the same arguments as in section 3 the insertion of the endpoints in the underlaying structure and the possible rebuilding of subtrees take amortized time $O(\frac{P_B(n)}{n}\log^2 n)$ per insertion. Inserting the segment in the at most $O(\log n)$ B-structures takes time $O(I_B(n)\log n)$. Inserting the segment in the at most 2 segment trees takes time $O(\log n)$. The bound for deletions follows in the same way. □

Most queries on segment partition trees take time $O(n^{\log_2(1+\sqrt{5})-1}Q_B(n))$ because all structures associated with nodes whose range is intersected by a particular line are queried. Replacing the conjugation tree by a δ-conjugation tree, such queries will take time $O(n^{\log_2(1+\sqrt{5})-1+\gamma}Q_B(n))$ for arbitrary small $\gamma > 0$.

Let us demonstrate this with one type of queries treated in [14]: Let S be a set of non-intersecting line segments, determine those segments that intersect a particular query line segment s. In this case all segments that have to be stored at a particular node v of the segment partition tree fully intersect ran_v and do not intersect each other. Note that they all intersect the edge of $bound_v$ belonging to the splitting line $l_{father(v)}$ at the father of v. We simply store the segments ordered by intersection with this edge in a tree. So B is a simple binary tree. Those segments at node v intersecting the query segment s are located by binary search on B (plus a stabbing query on S, see [14]) in time $O(\log n)$ plus the number of answers found. So $Q_B(n) = O(\log n)$, $I_B(n) = D_B(n) = O(\log n)$, $P_B(n) = O(n \log n)$ and $M_B(n) = O(n)$. Applying the dynamization method we obtain the following result:

Theorem 5.3 *Let S be a set of n non-intersecting line segments in the plane. Then for every $\gamma > 0$ there exists a data structure for storing S which uses $O(n \log n)$ storage and which can be constructed in $O(n \log^2 n)$ time such that line segment intersection queries can be performed in time $O(n^{\log_2(1+\sqrt{5})-1+\gamma} + k)$, where k is the number of reported answers, and updates can be performed in $O(\log^3 n)$ amortized time.*

Proof. The preprocessing time, amount of storage required and update time follow from lemma 5.1 and 5.2. The query time bound follows in the following way: For each γ' there exists a δ such that in a δ-conjugation tree each line l passes through the ranges for at most $O(n^{\log_2(1+\sqrt{5})-1+\gamma'})$ nodes. Clearly, only associated structures in nodes for which the query segment s passes through the range have to be checked. Hence, the query time will be $O(n^{\log_2(1+\sqrt{5})-1+\gamma'} \log n)$. Now choose γ' such that $0 < \gamma' < \gamma$ and we obtain the desired result. □

By constructing the tree in a slightly more careful way, the preprocessing time can be reduced to $O(n \log n)$ (see [14]). In this way the update time is reduced to $O(\log^2 n)$. Similar results can be obtained for the other types of queries treated in [14], like e.g. triangle stabbing queries.

A second possibility is to apply the techniques for decomposable searching problems to segment partition trees. To this end we have to show how weak deletions can be performed on segment partition trees. This is rather easy. We simply remove the segment s to be deleted from all associated B- and S-structures it belongs to. These structures can be determined in time $O(\log n)$ and the actual deletions will take time $O(D_B(n) \log n)$. Now applying Theorem 7.3.3.2 of [12] we obtain

Theorem 5.4 *Let PR be a decomposable searching problem that can be solved using a segment partition tree with query time $Q(n)$. Then there exist a dynamic structure for solving PR with the following performances:*

storage	$O(M_B(n)\log n)$
preprocessing	$O(P_B(n)\log n)$
query time	$O(Q(n))$
insertion time	$O(\frac{P_B(n)}{n}\log^2 n)$
deletion time	$O(\frac{P_B(n)}{n}\log n + D_B(n)\log n)$

These are worst-case update time bounds. We can apply this to the segment intersection query problem which is obviously decomposable.

Theorem 5.5 *Let S be a set of n non-intersecting line segments in the plane. Then there exists a data structure for storing S which uses $O(n\log n)$ storage and which can be constructed in $O(n\log^2 n)$ time such that line segment intersection queries can be performed in time $O(n^{\log_2(1+\sqrt{5})-1}\log n+k)$, where k is the number of reported answers, and insertions can be performed in time $O(\log^3 n)$ and deletions in time $O(\log^2 n)$.*

Again we can reduce the preprocessing time to $O(n\log n)$ which improves the insertion time to $O(\log^2 n)$.

6 Interval partition trees

The second type of structure for storing line segments defined by Overmars, Schipper and Sharir [14] is the *interval partition tree.* Interval partition trees have also a conjugation tree $\mathcal{T}$ on the set of endpoints of the segments as main structure and nodes contain associated structures, storing subsets of the segments. But this time segments are stored in a different way. Each segment s is stored at most twice in the tree. We determine the highest node v such that s intersects l_v. If s lies on l_v we store s in an ordinary interval tree S_v associated with v. (See e.g. [5, 16] for a description of interval trees.) Otherwise we store at both sons the part of s that lies inside its range in a B-structure, depending on the type of problem. The advantage over segment partition trees is that the amount of storage used will be reduced. Unfortunately, the structure can not be used for all searching problems. In [14] the following lemma is shown:

Lemma 6.1 *An interval partition tree $\mathcal{T}$ uses $O(M_B(n))$ storage and can be constructed in time $O(P_B(n))$.*

The structure can again be dynamized using partial rebuilding in exactly the same way as for segment partition trees. We replace the conjugation tree $\mathcal{T}$ by a δ-conjugation tree and perform an update by inserting or deleting the segment from the appropriate B- or S-structures, inserting or deleting the endpoints and rebuilding the subtree that gets out of balance. This immediately leads to the following result:

Lemma 6.2 *Insertions in an interval partition tree can be performed in* $O(I_B(n) + \frac{P_B(n)}{n} \log n)$ *amortized time and deletions in* $O(D_B(n) + \frac{P_B(n)}{n} \log n)$ *amortized time.*

For most types of queries on interval partition trees the query time becomes $O(n^{\log_2(1+\sqrt{5})-1+\gamma} Q_B(n))$ for arbitrary small $\gamma > 0$.

Also decomposability can be used. This easily leads to the following result:

Theorem 6.3 *Let PR be a decomposable searching problem that can be solved using an interval partition tree with query time $Q(n)$. Then there exist a dynamic structure for solving PR with the following performances:*

storage	$O(M_B(n))$
preprocessing	$O(P_B(n))$
query time	$O(Q(n))$
insertion time	$O(\frac{P_B(n)}{n} \log n)$
deletion time	$O(\frac{P_B(n)}{n} + D_B(n))$

In [14] the interval partition tree is used to solve the shooting problem. Unfortunately, the associated B-structure used for solving this problem is not dynamic and, hence, no good deletion time can be obtained.

7 Hidden surface removal

In this section we will apply the dynamic segment partition tree to solve the hidden-surface removal problem: Given a set V of triangles in space, compute which parts of the triangles are visible when looking from a particular viewpoint. Assuming that there is no cyclic overlap among the triangles, a simple transformation turns the problem into an instance where the triangles lie parallel to the xy-plane and the viewpoint lies at $z = -\infty$.

The basic idea behind the method is the following: We add the triangles one by one, starting at the bottommost triangle. At each step we keep track of the contour $\mathcal{C}$ of the triangles treated so far. For the next triangle t we compute the parts outside $\mathcal{C}$, which are visible and can be reported. Next we update the contour. See figure 1 for an illustration.

So we need a dynamic structure for maintaining the contour. The contour $\mathcal{C}$ consists of a number of polygonal regions, possibly with holes. The boundary of the contour consists of a set of non-intersecting line segments. These line segments we store in the dynamic segment partition tree $\mathcal{T}$ used in theorem 5.5 such that we can efficient determine the segments intersected by the boundary of the triangle t. We also store the vertices of the contour in a dynamic structure $\mathcal{T}'$ for half planar range searching as given in theorem 4.3. This structure can also efficiently determine those points that lie inside a given triangle. Finally we store $\mathcal{C}$ as a dynamic structure $\mathcal{P}$ for planar point location as described in Overmars [13] (or Preparata and Tamassia [17])

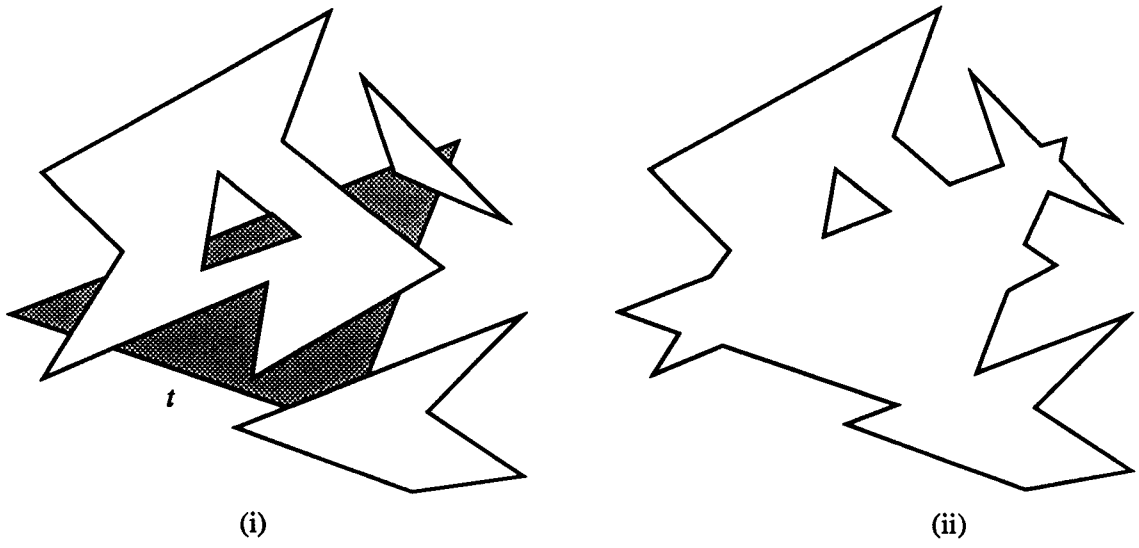

Figure 1: (i) determine the parts of t outside the contour and (ii) update the contour.

that allows for updates in time $O(\log^2 n)$ while we can determine in time $O(\log^2 n)$ whether a given point lies inside the contour or not.

In the beginning all these structures are empty. Let t be the next triangle to treat. For each edge s of the triangle we use the structure $\mathcal{T}$ to determine the intersections of s with the contour. From the list of intersections we can easily decide which parts of s are visible. If s does not intersect any edge of the contour, we search with one of its endpoints in $\mathcal{P}$ to see whether s lies completely inside the contour (in which case it is not visible) or completely outside. In this way we find all parts of the boundary of t that are visible.

To update the structures we proceed as follows. All parts of the edges that are visible must be inserted in $\mathcal{T}$ and $\mathcal{P}$ and their endpoints in $\mathcal{T}'$. From the query we also find those edges of the contour that are intersected by the triangle boundary. These must be shortened to the piece(s) outside t. This can be done by deleting them and inserting the reduced pieces. Finally we have to remove those edges of the contour that lie completely inside t. To this end we perform a query with t on $\mathcal{T}'$ to find the vertices inside t and, hence, the edges inside t. We simply delete these edges from all the structures. It can easily be seen that in this way the contour is correctly updated.

Theorem 7.1 *The hidden surface removal problem for a set of n triangles (without cyclic overlap) can be solved in time $O(n \log n + n \cdot c^{\log_2(1+\sqrt{5})-1} \log c + k \log^2 c)$ where k is the output size and c the maximal contour size during the construction.*

Proof. Sorting the triangles by depth will take time $O(n \log n)$. The size of all the data structures is bounded by c. On each structure we perform $O(n)$ queries.

The most expensive structure is the segment partition tree which has a query time of $O(c^{\log_2(1+\sqrt{5})-1} \log c + k')$. Each reported intersection (in $\mathcal{T}$) or point (in $\mathcal{T}'$) can be charged to part of the output visibility map. Hence, the total amount of time spend for queries is $O(n \cdot c^{\log_2(1+\sqrt{5})-1} \log c + k)$ where k is the output size.

The total number of updates on the structures is $O(k)$. The update time on each of the structures is bounded by $O(\log^2 c)$. Hence, the total time spend on updates in $O(k \log^2 c)$. □

The time bound can be restated as $O(n \cdot k^{\log_2(1+\sqrt{5})-1} \log n)$ because $c \leq k$ and $k \log^2 c = O(n \cdot k^{\log_2(1+\sqrt{5})-1} \log n)$ (because $k = O(n^2)$).

8 Concluding remarks

In this paper two dynamic versions of conjugation trees have been presented. The first version slightly relaxes the balance condition of the structure and uses partial rebuilding to perform updates in $O(\log^2 n)$ time. The second technique uses the decomposability of most of the searching problems for which partition trees are used and applies standard techniques from [12]. Both methods also apply to two related data structures, segment partition trees and interval partition trees, that store line segments and answer various types of queries on them. As an application we obtained an output-sensitive method for hidden surface removal. (Very recently some new, more efficient methods have been designed, see [15, 18, 19].)

Some open problems do remain. In particular, it would interesting to see in which way the techniques presented here can be applied to other partition trees. Using decomposability will in general be possible (although it is not always clear how to perform weak deletions). Partial rebuilding on the other hand requires a special type of balance criteria and a method for efficiently rebuilding subtrees that do not affect the total tree. Especially the partition trees of Haussler and Welzl [9] form an interesting candidate for this research.

References

[1] Agarwal, P.K., A deterministic algorithm for partitioning arrangements of lines and its applications, *Proc. 5th ACM Symp. on Computational Geometry*, 1989, pp. 11–22.

[2] Bentley, J.L., Decomposable searching problems, *Inform. Proc. Letters* **8** (1979), 244–251.

[3] Chazelle, B., Polytope range searching and integral geometry, *Proc. 28th Symp. on Foundations of Computer Science,* 1987, pp. 1–10.

[4] McCreight, E.M., Priority search trees, *SIAM J. Computing* **14** (1985), 257–276.

[5] Edelsbrunner, H., Intersection problems in computational geometry, Forschungsberichte F93, Inst. f. Informationsverarbeitung, TU Graz, 1982.

[6] Edelsbrunner, H., *Algorithms in combinatorial geometry,* Springer-Verlag, Berlin, 1987.

[7] Edelsbrunner, H. and E. Welzl, Halfplanar range search in linear space and $O(n^{0.695})$ query time, Forschungsberichte F111, Inst. f. Informationsverarbeitung, TU Graz, 1983.

[8] Edelsbrunner, H. and E. Welzl, Halfplanar range search in linear space and $O(n^{0.695})$ query time, *Inform. Proc. Letters* **23** (1986), 289–293.

[9] Haussler, D. and E. Welzl, Epsilon nets and simplex range queries, *Discrete Computational Geometry* **2** (1987), 127–151.

[10] Matoušek, J., Construction of ϵ-nets, *Proc. 5th ACM Symp. on Computational Geometry*, 1989, pp. 1–10.

[11] Megiddo, N., Partitioning with two lines in the plane, *J. Algorithms* **6** (1985), 430–433.

[12] Overmars, M.H., *The design of dynamic data structures,* Springer-Verlag, Berlin, 1983.

[13] Overmars, M.H., Range searching in a set of line segments, *Proc. 1st ACM Symp. on Computational Geometry*, 1985, pp. 177–185.

[14] Overmars, M.H., H. Schipper and M. Sharir, Storing line segments in partition trees, Techn. Rep. RUU-CS-89-17, Dept. of Computer Science, Utrecht University, 1989.

[15] Overmars, M.H., and M. Sharir, Output-sensitive hidden surface removal, *Proc. 30th IEEE Symp. on Foundations of Computer Science*, 1989, pp. 598–603.

[16] Preparata, F.P., and M.I. Shamos, *Computational geometry: An introduction,* Springer-Verlag, Berlin, 1985.

[17] Preparata, F.P., and R. Tamassia, Dynamic techniques for point location and transitive closure in planar structures, *Proc. 29th IEEE Symp. on Foundations of Computer Science*, 1988, pp. 558–567.

[18] Sharir, M., and M.H. Overmars, A simple output-sensitive algorithm for hidden surface removal, Techn. Rep. RUU-CS-89-26, Dept. of Computer Science, Utrecht University, 1989.

[19] Sharir, M., and M.H. Overmars, An improved technique for output-sensitive hidden surface removal, Techn. Rep. RUU-CS-89-32, Dept. of Computer Science, Utrecht University, 1989.

[20] Welzl, E., Partition trees for counting and other range searching problems, *Proc. 4th ACM Symp. on Computational Geometry*, 1988, pp. 23–33.

[21] Willard, D.E., Polygon retrieval, *SIAM J. Computing* **11** (1982), 149–165.