DYNAMIC ENVIRONMENTAL MODELING BY THE C–TREE

KNUT VERBARG*
Department of Computer Science, University of Würzburg
Am Hubland, D-97074 Würzburg, Germany

and

HARTMUT NOLTEMEIER*
Department of Computer Science, University of Würzburg
Am Hubland, D-97074 Würzburg, Germany

ABSTRACT

We introduce an efficient and robust spatial index to support a set of different queries, which is developed from Günther's *Celltree*[6] and the *Monotonous Bisector* Tree*[11,16]. In practice, huge scenes are manageable by using the *paging–concept*. For convex polyhedrons in N–dimensional real space and any L_p–metric ($1 \leq p \leq \infty$) we are able to show that the C-tree can be constructed in time $\mathcal{O}(n \log n)$, linear space and logarithmic height, where n denotes the number of objects. Dynamic insertion of objects is performed in $\mathcal{O}(\log^2 n)$ amortized worst–case time. Objects can be deleted in amortized $\mathcal{O}(n)$ or in amortized $\mathcal{O}(\log^2 n)$ if the update of cluster radii can be delayed. In all cases logarithmic height and linear space requirements are preserved.

1. Introduction

In a wide range of applications the crucial problem is to represent the neighborhood structures in huge scenes of spatial objects in an adequate and efficient manner. One of these applications is environmental modeling in robotics. Here, we must be able to dynamically update the known scene due to sensory input and to support fast access to the objects through their spatial attributes, e.g. querying the local scene or plan a collision–free motion. Moreover in very large or detailed scenes it is necessary to store the objects in the slower secondary memory also. The model of *Paging* is

*email: `[verbarg,noltemei]@informatik.uni-wuerzburg.de`

used to describe this physical restriction. The data structure has to be partitioned into pages which correspond to memory units. The efficiency criterion is the number of page accesses needed.

In the past, numerous spatial indexes coping with these requirements have been developed[6,10,12].

One example is the *Celltree*[6] introduced by Günther. The Celltree is a dynamic rooted tree. It is designed for secondary memory, i.e. the nodes can be paged. Its basic shape is that of the *B–tree*[2], one of the classical tree structures for secondary memory. Each node represents a cell of the space of objects. It is partitioned completely into disjoint cells, each of which is the cell of one of its successors. The subsets are obtained by *binary space partitioning (BSP)*[4], i.e. the cell decomposition of each node is described by a BSP–tree. The assignment of objects to a cell may follow one of the well–known conflict strategies for extended objects[14].

The Celltree is affected with many open parameters and problems. In the following, we replace the BSP–tree by an improved alternative: the *Monotonous Bisector* Tree*[16,11]. This results in a new structure, the *C-tree*, that combines the advantages of the Celltree and the Monotonous Bisector* Tree.

To speed up searching, Günther proposes containers to give a tighter approximation of the objects stored in a subtree than the given partitions do. By using a Monotonous Bisector* Tree as BSP such a container is automatically realized as a sphere of the cluster center with cluster radius.

With the knowledge of the Monotonous Bisector* Tree we are able to use the balancing step instead of heuristically splitting overflowing pages with a plane sweep in l directions. This guarantees a partition into equally sized halves and is even applicable in $\mathbb{R}^N$. The Günther–Celltree demands at least m sons for each internal node. But without a rebalancing strategy this rigid shape can not be guaranteed, since splitting may fail and cause overflow pages.

The remainder is organized as follows. Sec. 2 contains the definitions of the C-tree structures. In Sec. 3, an algorithm is presented creating C-trees within $\mathcal{O}(n \log n)$ time, where n is the number of objects to be inserted. The queries efficiently supported by the C-tree are compiled in Sec. 4. In Sec. 5 we show that objects can be inserted in $\mathcal{O}(\log^2 n)$ amortized worst–case time. Deletion of objects can be performed in $\mathcal{O}(n)$ amortized time, and in $\mathcal{O}(\log^2 n)$ amortized time, if the update of the so–called cluster radii can be delayed.

2. The C-tree

A Monotonous Bisector* Tree is a binary leaf–oriented rooted tree to represent a set S of arbitrary objects (not necessarily of the $\mathbb{R}^N$). The objects are only clustered according to their neighborship relation, modeled by a distance function. The partitioning of the object space is thereby described by a set E of simple auxiliary objects. Therefore this concept is superior to common spatial indices, because no embedding in the $\mathbb{R}^N$ is necessary. Only the topology of the objects space must be modeled by a distance function. This allows to handle more abstract (not geometric) or only partially geometrically described problems.

E is now a set of meaningful auxiliary points to create the directory. The only restriction on E is that an efficiently computable distance function $d : E \times S \to \mathbb{R}_{\geq 0}$ must be available. The idea is to keep E simple, such that the computation of d is cheaper than computing the distance between two complex objects of S. On the other hand E must be chosen powerful enough to allow a flexible clustering.

A N–dimensional sphere $K(v)$ (induced by the metric d) is assigned to each node v in the tree. $K(v)$ is described by a cluster center (split value) and a cluster radius. The cluster of a node is the finite intersection

$$\mathrm{Cluster}(v) := \cap\{K(\tilde{v}) \mid \tilde{v} \text{ lies on the path from the root to } v\}.$$

In general, the partitioning is neither disjoint nor complete. The conflicts are resolved with the strategy overlapping clusters in the following way: An object is assigned to the cluster with the nearest cluster center. If the decision is not unique, any of the nearest clusters is chosen.

Thus, objects are partitioned with *bisectors*. The bisector of $e_1, e_2 \in E$ is defined as the set of all $s \in S$ with $d(e_1, s) = d(e_2, s)$. In the case of $E = \mathbb{R}^2$, $S \subset \mathcal{P}(\mathbb{R}^2)$ and the Euclidean distance d the bisector is the mid–perpendicular of e_1 and e_2.

If S is a set of extended objects in $\mathbb{R}^N$, simple examples show that it is sometimes impossible to create a balanced tree with $S = E$. So the augmention with artificial split values is meaningful, if it is applicable to the used space.

Let S be the set of objects, E the set of adequate splitting objects and $d : E \times S \mapsto \mathbb{R}_{\geq 0}$ the corresponding distance function, which is efficiently computable.

Definition 1: A *Monotonous Bisector* Tree (MBT)* is an ordered binary rooted tree with the set of marked nodes $V \cap \mathbb{N}_0 = \emptyset$ in the following way:

1. To each node $v \in V$ there is related an 8–tuple of the form[a]

$$(\mathrm{lSplit}(v),\ \mathrm{lRadius}(v),\ \mathrm{lSon}(v),\ \mathrm{lObject}(v),$$

$$\mathrm{rSplit}(v),\ \mathrm{rRadius}(v),\ \mathrm{rSon}(v),\ \mathrm{rObject}(v))$$

   with $@\mathrm{Split}(v) \in E$, $@\mathrm{Radius}(v) \in \mathbb{R}_{\geq 0}$, $@\mathrm{Son}(v) \in V \cup \mathbb{N}_0 \cup \{\mathsf{nil}\}$, $@\mathrm{Object}(v) \subset S$.

2. Tree structure:
   An implicit representation of the tree structure is given through

$$\mathrm{lSon}(v) \begin{cases} \in \mathbb{N}_0 \cup \{\mathsf{nil}\}, & \text{if } v \text{ has no left successor} \\ \in V, & \text{namely the left successor of } v\text{, otherwise} \end{cases}$$

   (analogously for $\mathrm{rSon}(v)$).

[a]Always textually replace @ by the letter `'l'` and `'r'`. In the following we will use this shorter notation.

3. Objects are stored in leaves only:
   If $@\mathrm{Son}(v) \neq \mathsf{nil}$ for $v \in V$, then $@\mathrm{Object}(v) = \emptyset$.

4. Heredity of split values:
   For all $v \in V$ with $@\mathrm{Son}(v) =: \tilde{v} \in V$ we have $@\mathrm{Split}(v) = \mathrm{lSplit}(\tilde{v})$.

5. Numbering of the partitions:
   For all $v, \tilde{v} \in V$ with $@\mathrm{Son}(v), \tilde{@}\mathrm{Son}(\tilde{v}) \neq \mathsf{nil}$ teh condition $@\mathrm{Son}(v) \neq \tilde{@}\mathrm{Son}(\tilde{v})$ holds.

If $\mathrm{lSon}(v)$ or $\mathrm{rSon}(v) \in V$ respectively, then they point to the sons of v in MBT. If they are in $\mathbb{N}_0$, then v does not have a left (or right) son. In this case they contain the number of the next node in the C-tree, which is related to their partition. Finally if they are nil, then $\mathrm{lObject}(v)$ (or $\mathrm{rObject}(v)$) is the set of represented objects in this partition. At this point only the structure of the MBT is defined, but not the corresponding semantics (the way objects are assigned to partitions and the form of the partitions). This will be done in the sequel.

The MBT is mainly used for describing the partitions in the C-tree, since the objects should be stored as deep as possible (in the leaves) in the C-tree.[b] The demand to have an 8–tuple for each node in the MBT is not a restriction. An empty tree can be constructed by choosing an artificial split value for rSplit(root) and setting rSon(root):= nil, rObject(root) := $\emptyset$.

For the sake of a simpler notation we give the following two definitions.

Definition 2: For $e \in E$ and $r \in \mathbb{R}_{\geq 0}$ we define $\mathit{Sphere}(e, r) := \{s \in S \mid d(s, e) \leq r\}$. $e \in E$ is called a *split value* or a *cluster center*.

Definition 3: For a MBT with nodes V we define:

1. $v \in V$ is called *@Partition*, if and only if $@\mathrm{Son}(v) \in \mathbb{N}_0$.

2. $v \in V$ is called *@Leaf*, if and only if $@\mathrm{Son}(v) = \mathsf{nil}$.

To be able to page objects, they should fit on a page of bounded size. Therefore we specify the objects manageable by the C-tree. Originally only convex polygons could be represented. This is where the term "cell" comes from. Let $k \in \mathbb{N}$ be given:

Definition 4: A *cell* is an object, that can be described with k real parameters using a fixed modeling.

[b] In the Günther–Celltree originally there was a sharp distinction between nodes to store objects (the leaves) and directory nodes (internal nodes). The in fact growing main memories provide the design of larger pages. In this approach this would force large pages for objects. These would not be able to be very distinctive any more. We avoid this problem through mixing directory and objects on one page. Furthermore underflowing object buckets can be handled cheaper.

We can model for example all convex polygons with at most $\lfloor \frac{k}{2} \rfloor$ vertices in $\mathbb{R}^2$ by the convex hull of their vertices. The type of modeling is arbitrary, but equal for all objects. In fact, there may be only a unique key for each object. Only the distance function uses the modeling. For realizing the C-tree it is only important to be able to store the objects.

Let S be such a universe of cells. We will now define the C-tree.

Definition 5: The C-tree (CT) for the parameter $P \in \mathbb{N}$ is a leaf–oriented rooted tree with nodes $V = \{0, 1, \ldots, N\}$ to represent a set $C \subset S$ of cells, with the following properties:

1. Paging

   (P) Each node fits on a page of capacity P in the secondary memory.

2. Structure:

   (S1) Each node $v \in V$ represents a MBT(v) with nodes $V(v)$.

   (S2) The edges of the C-tree are implicitly given by
   $v_{\mathrm{CT}}{}'$ is father of v_{CT} if and only if
   $\exists v_{\mathrm{MBT}} \in V(v_{\mathrm{CT}}{}') : v_{\mathrm{MBT}}$ is @Partition in MBT($v_{\mathrm{CT}}{}'$) and @Son(v_{MBT}) $= v_{\mathrm{CT}}$.

   (S3) Heredity of split values:
   If $v_{\mathrm{CT}}{}'$ is father of v_{CT} and $v_{\mathrm{MBT}} \in V(v_{\mathrm{CT}}{}')$ with @Son(v_{MBT}) $= v_{\mathrm{CT}}$, then @Split(v_{MBT}) = the left split value of the root of MBT(v_{CT}).

   For $v_{\mathrm{CT}} \in V$ and $v_{\mathrm{MBT}} \in V(v_{\mathrm{CT}})$ we denote with $w(v_{\mathrm{MBT}})$ the unique path consisting of MBT–nodes starting from the root of the C-tree to v_{MBT}, and with $W(v_{\mathrm{CT}})$ the corresponding path in the C-tree.

3. Representation of objects:
   For $v_{\mathrm{CT}} \in V$ and $v_{\mathrm{MBT}} \in V(v_{\mathrm{CT}})$ we term
   $@\mathit{Cluster}(v_{\mathrm{MBT}}) := \mathrm{Sphere}(@\mathrm{Split}(v), @\mathrm{Radius}(v)) \cap$
   $\bigcap \ \{\mathrm{Sphere}(\tilde{@}\mathrm{Split}(\tilde{v}), \tilde{@}\mathrm{Radius}(\tilde{v})) \mid \tilde{v}, \tilde{@}\mathrm{Son}(\tilde{v}) \in w(v_{\mathrm{MBT}})\}$.

   (O1) Cluster cover cells:
   For all $v_{\mathrm{CT}} \in V$ and $v_{\mathrm{MBT}} \in V(v_{\mathrm{CT}})$ it is @Object(v_{MBT}) $\subset$ @Cluster(v_{MBT}).

   (O2) Unique search path:
   For $v_{\mathrm{CT}} \in V$ and $v_{\mathrm{MBT}} \in V(v_{\mathrm{CT}})$ it is $c \in$ @Object(v_{MBT}) if and only if:
   $c \in C$ and $\forall v_{\mathrm{MBT}}{}' \in w(v_{\mathrm{MBT}}) \, \forall v_{\mathrm{CT}}{}' \in W(v_{\mathrm{CT}})$ holds:
   $d(\mathrm{lSplit}(v_{\mathrm{MBT}}{}'), c) < d(\mathrm{rSplit}(v_{\mathrm{MBT}}{}'), c)$
   $\Leftrightarrow \quad \mathrm{lSon}(v_{\mathrm{MBT}}{}') \in w(v_{\mathrm{MBT}}) \cup W(v_{\mathrm{CT}})$ or $v_{\mathrm{MBT}}{}' = v_{\mathrm{MBT}}, @ = \texttt{'l'}$.

The C-tree is a rooted tree of MBTs, describing the paging of the virtual MBT. The defined path $w(v_{\mathrm{MBT}})$ is related to this virtual MBT. Furthermore the cluster of a node v_{MBT} is the intersection of all spheres appearing on the path from the root to

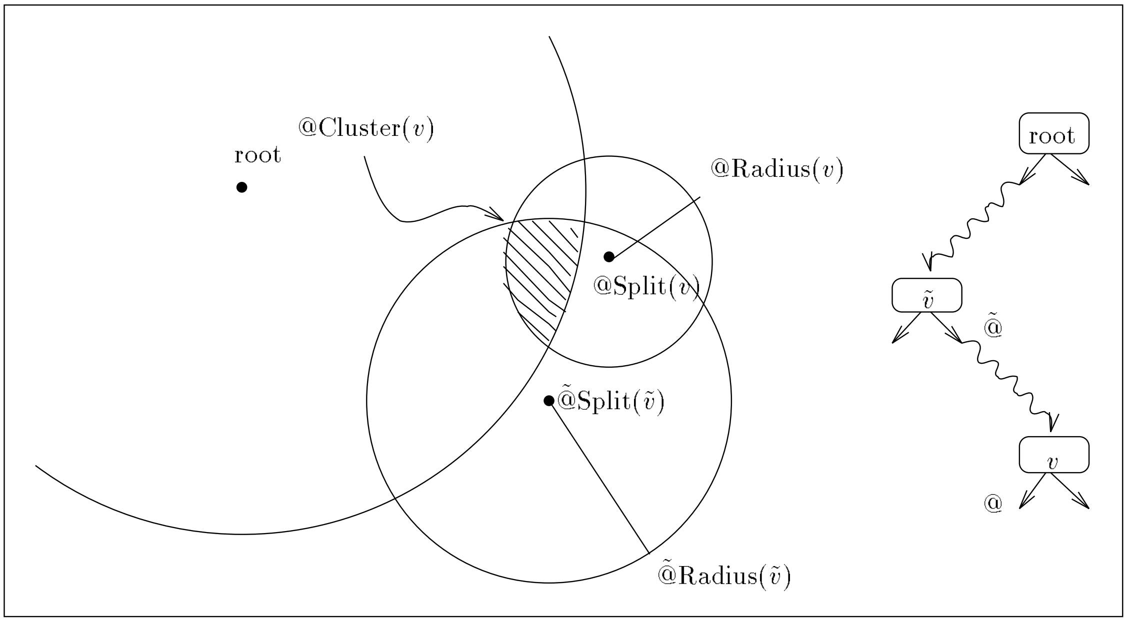

Fig. 1. The Cluster of a node.

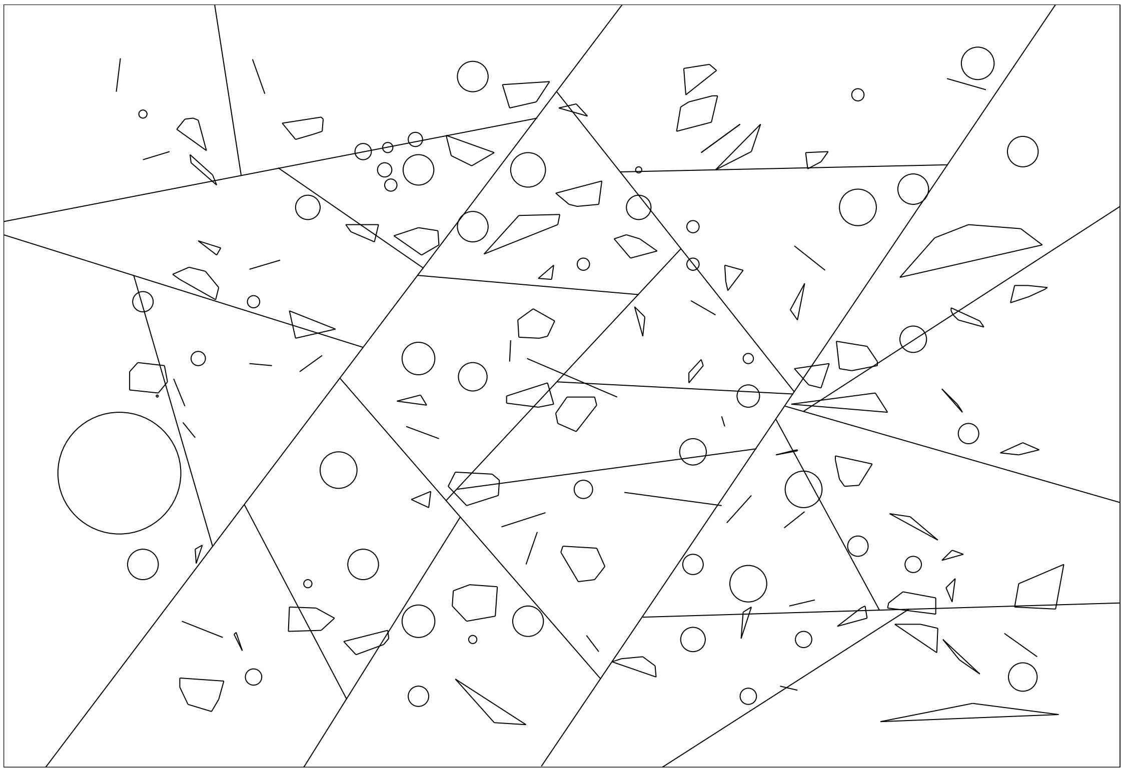

Fig. 2. Partitioning of a scene through the root of the C-tree.

v_{MBT} in the virtual MBT. This is shown in Fig. 1. (O2) expresses that one has to go "left" (or "right" respectively) on a search path, if the left (right) split value is nearer to the considered object.

Fig. 2 shows for a scene of convex objects the MBT of the root of the C-tree. For clearance subsequently the partitioning of the scene is represented by relevant parts of the bisector of adjacent split values (compare the Voronoi diagram). The MBT is partitioning the plane into clusters of convex polygons. The large degrees of freedom in choosing the MBT for a node of the C-tree provides a good balancing of the tree. The next section shows, how this can be done.

Let sizeof(cell) be the space requirement of a cell and sizeof(mbt_node) of a MBT–node. According to (P) for a fixed page size P the maximum number of sons of a node in the C-tree is $M := \left\lfloor \frac{P}{\text{sizeof(mbt\_node)}} \right\rfloor + 1$ and $\bar{M} := \left\lfloor \frac{P - \text{sizeof(mbt\_node)}}{\text{sizeof(cell)}} \right\rfloor$ is the maximum number of cells in a node.[c] Therefore a C-tree of height h contains at most $\bar{M}M^h$ cells.

3. Creating the C-tree with Estimated Height, Effort and Space Requirements

The idea is to create a Monotonous Bisector* Tree and to map it to the C-tree, i.e. to page it. To do this, recursively a second split value belonging to the heredited split value is chosen. The aim is to partition the set $C \subset S$ into two equal parts and decrease the cluster radii as fast as possible. This is achieved by two algorithms which can be applied flexibly. The partitioning with bisectors automatically makes the Monotonous Bisector* Tree flexible in respect to the distance function d.

The *balancing step* chooses for a given $C \subset S$ and $e_1 \in E$ a second split value e_2, such that both resulting clusters C_1 and C_2 contain at least a linear portion of C. This is very tricky, but possible in linear time $\mathcal{O}(|C|)$ for a set S of compact objects in $\mathbb{R}^N$, $E = \mathbb{R}^N$ and d a Minkowski–metric[16]. This type of partition step guarantees the balancing of the tree and therefore bounds the height of the tree logarithmically.

The *contraction step* chooses e_2 such that the cluster radii decrease as fast as possible. Let $o_1, \ldots, o_k \in C$ be the objects with the same maximum distance of e_1. Let O be a point on any of the o_i, $(i = 1, \ldots, k)$, taking on the distance. Now e_2 is chosen carefully to get as many objects as possible (but at least O) into its cluster. In the worst case only O is separated from the original cluster and the cluster radius is not necessarily decreasing. Nevertheless it is possible to reduce the cluster radius to any fraction $q \in]0; 1[$ with a fixed number $l = l(q)$ of contraction steps. These steps do not have to be consecutive. Thereby the number $l \in \mathbb{N}$ of steps depends on q and the space S (in $\mathbb{R}^N$ especially on the dimension[16]). In our implementation we used $e_2 := \frac{2}{3}O + \frac{1}{3}e_1$. This step also costs $\mathcal{O}(|C|)$ time, since only the distance to each object is computed.

Experimental results show that both the balancing step and the contraction step

[c] Each page contains at least one MBT–node.

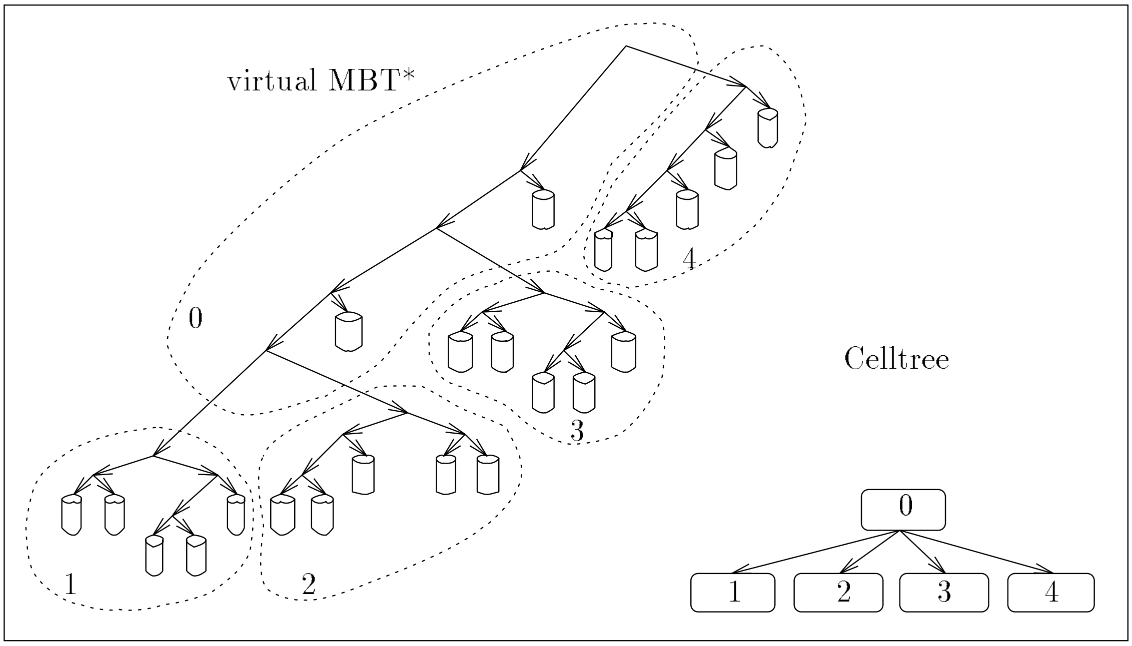

Fig. 3. Paging the virtual Monotonous Bisector* Tree onto the C-tree.

are necessary to achieve a "good" tree. But the latter does not allow a statement concerning the balance. Therefore the rigid demands for the shape of the Günther–Celltree[d] can not be realized. This is the reason why the structure of the C-tree in Sec. 2 is designed to be more flexible.

The paging raises additional demands to the creation process. Paging is only possible if the creation is feasible with bounded main memory, too. We can state that the balancing and contraction steps are executable under these restrictions, if the objects are given in a file.

If the page size P becomes too large, it is not sensible to store up to $\bar{M}$ objects in a leaf. So we introduce a bucket size B. Every bucket in the C-tree has got at most B objects: $\forall v \in V \quad \forall v' \in V(v) \quad \forall @ \in \{'\mathtt{l}', '\mathtt{r}'\} : |@\mathrm{Object}(v')| \leq B$.

The creation algorithm partitions hierarchically the set of objects using balancing and contraction step until a CT–node is filled. It then proceeds recursively with the sons. Thereby the following strategy is used: *For the sake of balanced subtrees, always the largest cluster is partitioned next.* Fig. 3 shows how the successively created Monotonous Bisector* Tree is paged onto the C-tree.

[d]Shape of the Günther–Celltree:

- All leaves are on the same level.
- The root has got no or more than two sons.
- Each internal node except the root has got at least m sons, with m some constant.

Algorithm 1: Create (w, C, e, α, B)

w *: root of the C-tree*

$C \subset S$*: the set of objects to represent*

e *: heredited split value*

$\alpha \in [\frac{1}{2}; 1]$*: filling factor for internal nodes*

$B \leq \bar{M}$*: bucket size*

1. IF $|C| \leq B$, THEN *create a tree MBT(w) with only one node, which represents the objects in* C.

2. *Execute a partition step on* e *and* C*, which results in the second split value* e'.

   Let $V(w) := \{v\}$*, where* v *is a new MBT–node with: lSplit(v)* $:= e$*, rSplit(v)* $:= e'$*, lSon(v), rSon(v)* $:=$ nil *and lObject(v), rObject(v)* $:= \emptyset$.

   Split C *according to the split values into lSet(v) and rSet(v) and compute lRadius(v) and rRadius(v).*

3. WHILE $|V(w)| < \alpha M$ DO:

   Determine $(\tilde{v}, \tilde{@}) \in V_{leaf}$ *with* $|\tilde{@}Set(\tilde{v})| = \max\{|@Set(v)| \mid (v, @) \in V_{leaf}\}$.
   (The cluster with the largest cardinality is partitioned. *)*

   IF $|\tilde{@}Set(\tilde{v})| \leq B$, THEN GOTO *4.*

   Execute a partition step on $\tilde{@}Split(\tilde{v})$ *and* $\tilde{@}Set(\tilde{v})$*, which results in the second split value* e'.

   Let $V(w) := V(w) \cup \{v\}$*, where* v *is a new MBT–node with: lSplit(v)* $:= \tilde{@}Split(\tilde{v})$*, rSplit(v)* $:= e'$*, lSon(v), rSon(v)* $:=$ nil *and lObject(v), rObject(v)* $:= \emptyset$.

   Split $\tilde{@}Set(\tilde{v})$ *according to the split values into lSet(v) and rSet(v) and compute lRadius(v) and rRadius(v).* $\tilde{@}Set(\tilde{v}) = \emptyset$.

   Last, insert the new node v *through* $\tilde{@}Son(\tilde{v}) := v$.

   ENDWHILE

4. FOR ALL $(v, @) \in V_{leaf}$ *with* $@Set(v) \neq \emptyset$ LET *@Son(v) := a new node in the C-tree and execute Create (@Son(v), @Set(v), @Split(v),* α*, B).*

The subsets of objects, that are temporarily created by the algorithm, must be held in secondary memory, too. They are termed as lSet(v) resp. rSet(v) according to the related MBT–node v. The set of actual leaves in MBT(w) is called $V_{\text{leaf}} := \{(v, @) \in V(w) \times \{\texttt{'l'}, \texttt{'r'}\} \mid v \text{ is @Leaf }\}$. The parameter α is a load factor for internal nodes. It is left open here which value of α is best in the dynamic case. B denotes the size of all buckets in the tree.

Theorem 1: Algorithm 1 creates a C-tree with root w, which represents the elements in C. The objects are stored in leaves and the buckets fulfill the maximum load B.

We will now estimate the balance of the created tree. To this end we consider the internal structure of a node. Card(v) denotes the number of objects represented in the subtree with root v (v is a MBT–node or a CT–node). In other words Card(v) *is the cardinality of the cluster of* v.

The theory of the Monotonous Bisector* Tree[16] states that for the balancing step (BS) holds: $\mathrm{Card}(@\mathrm{Son}(v)) \geq \lfloor \frac{\mathrm{Card}(v)}{2} \rfloor$, if the cells are convex polygons in $\mathbb{R}^N$ and the distance function d is any L_p–metric. Unlikely for the contraction step (KS) no such statement can be made. Using the KS, one of the two sons may have less than half of the original cells. We term this kind of sons as *underfilled*.

While creating the tree we are able to choose sensitively (according to the development of cluster radii and cardinality) the BS or KS partition step. *Subsequently we assume that at most every second step*[e] *is a KS.* Test of the Monotonous Bisector* Tree[11] showed that this is quite efficient. We now ask for the fraction q of filled sons of a CT–node, that is the number of filled leaves in the MBT.

An underfilled son can only be created by a KS, so: #underfilled sons $\leq$ #KS. On the other hand every BS creates at least one new filled son: #filled sons $\geq$ #BS $+ 1$.

How large can q get in the worst–case? For this the MBT must be created by as many KS as possible, but only few BS. We assume that a KS has got an underfilled son. Then this path is not used furthermore in Algorithm 1. So the worst–case looks like shown in Fig. 4. Hence it holds for the height $2h$ and $2h-1$ of the worst–case:

$$\frac{q}{1-q} = \frac{\#\text{filled sons}}{\#\text{underfilled sons}} \geq \frac{\#\text{BS}+1}{\#\text{KS}} \geq \frac{2^0+2^1+\ldots+2^{h-1}+1}{2^0+2^1+\ldots+2^h} = \frac{2^h}{2^{h+1}-1} \geq \frac{1}{2}.$$

So $2q \geq 1-q$ and hence $q \geq \frac{1}{3}$.

Remark: On the average one can expect as much BS as KS and so $q \approx \frac{1}{2}$. Furthermore the KS is normally balanced, if the objects are distributed equally. In general we do have a trade–off between a larger q and a "better" tree (cluster radii descent faster). This can be controlled through the number of contraction steps used.

We now know that $v \in V$ has got at least $M_{\alpha q} := \lfloor(\lceil M\alpha\rceil + 1)q\rfloor$ filled sons (denoted now $V' \subset V$), which represent at most Card(v) objects. The creation procedure partitioned in each case the largest cluster, so it is:

$$\max_{v'\in V'} \mathrm{Card}(v') \leq 2 \min_{v'\in V'} \mathrm{Card}(v') \tag{1}$$

$$\max_{v'\in V'} \mathrm{Card}(v')(1 + \underbrace{\frac{1}{2}+\ldots+\frac{1}{2}}_{(M_{\alpha q}-1)\text{–times}}) \leq \mathrm{Card}(v)$$

---

[e]On a path from the root to a leaf at most every second

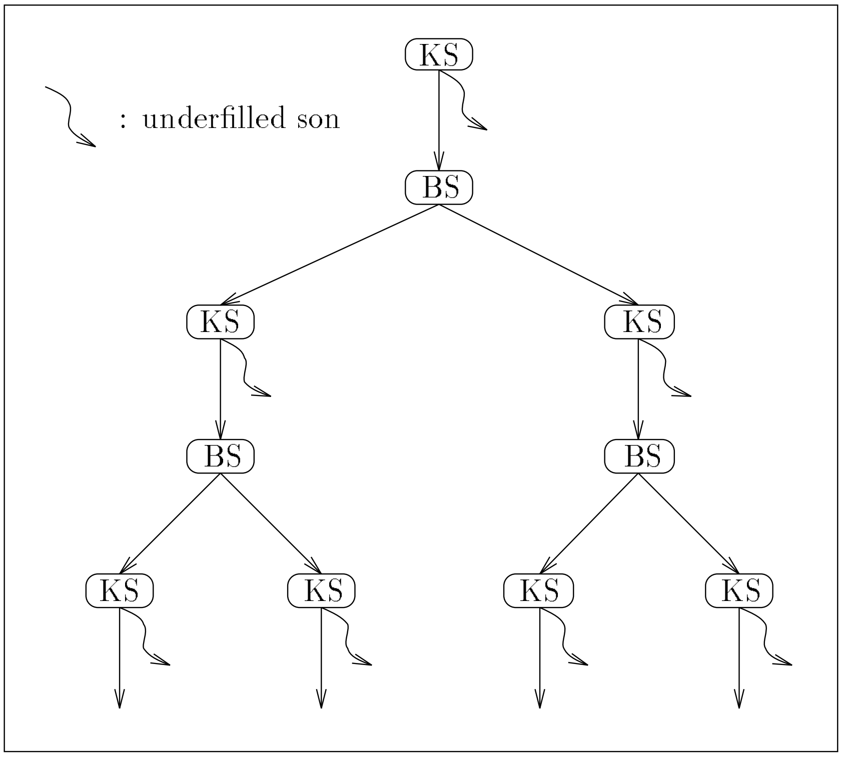

Fig. 4. Worst–case for q.

and hence:

$$\max_{v' \in V'} \mathrm{Card}(v') \leq \frac{2\mathrm{Card}(v)}{M_{\alpha q}+1}$$

This provides a guaranteed reduction factor for Card(v). With the help of this factor we are able to estimate the height of the tree:

Theorem 2: Let $M_{\alpha q} := \lfloor (\lceil M\alpha \rceil + 1) q \rfloor$, $q = \frac{1}{3}$, the cells be convex polygons of $\mathbb{R}^N$ and d any L_p–metric ($1 \leq p \leq \infty$). Then Algorithm 1 creates a C-tree with the height of at most

$$\begin{cases} \left\lceil \log_{\frac{M_{\alpha q}+1}{2}} \frac{|C|}{B} \right\rceil, \text{ if } |C| > B \\ 0, \text{ if } |C| \leq B \end{cases},$$

if at most each second step is not balanced and $M_{\alpha q} \geq 2$.

Theorem 3: Under the assumptions of Theorem 2, Algorithm 1 creates a C-tree in $\mathcal{O}(|C| \log |C|)$ time.

Proof: The virtual Monotonous Bisector* Tree can be constructed in $\mathcal{O}(|C| \log |C|)$ time: The balancing step provides logarithmic height. The total time directly results from this fact and the partitioning effort of $\mathcal{O}(|C|)$ in each step.

We will now show, that the additional time effort for paging the Monotonous Bisector* Tree is also $\mathcal{O}(|C|\log|C|)$. Therefore we organize V_{leaf} (the set of actual leaves) as a maximum–heap in respect to the cardinalities $|@\text{Set}(v)|$, $(v,@) \in V \times \{'\mathtt{l}','\mathtt{r}'\}$. The size of the heap is at most M. Hence deleting the maximum element and inserting needs $\mathcal{O}(\log M)$. Therefore step 3 causes an additional effort of $\mathcal{O}(M\log M) =$ constant, if M is fixed. For large page sizes P, also B and M may be large. But the theorem holds even if $M \in \mathcal{O}(|C|)$. □

Theorem 4: Under the assumptions of Theorem 2 the C-tree created by Algorithm 1 requires $\mathcal{O}(|C|)$ space (in secondary memory). More precisely the following bounds hold: For all internal nodes the load of pages is $\geq \alpha$, up to at most one exception on a path from the root to a leaf.

Proof: From step 3 of the algorithm we conclude that the iterated partitioning breaks off only if the load factor is fulfilled or each of the sons fits into one bucket. Therefore the load α is reached for all internal nodes except the last one on the path to a leaf. Since the Monotonous Bisector* Tree itself has got only linear size, the above load on an average shows that the space requirement is $\mathcal{O}(|C|)$. □

We now have reached the aim of an average page load. We have seen that like a rigid tree shape, the average load prevents from degenerating and is more flexible. Moreover on any search path to a leaf at most two underfilled nodes are visited. Therefore underfilled pages can not cumulate on a search path.

We can improve the bounds of the above theorems in the case $B \ll \bar{M}$, if we try to put more buckets in one page. More precisely from the guaranteed reduction factor in Theorem 2 we can derive a number $\bar{M}_B$, which is the maximum number of objects a C-tree with one node can be constructed for (step 1 in Algorithm 1). Instead of B in step 3, we have then the threshold $\bar{M}_B$. This saves the load factor for internal nodes and raises the load of leaves to $\frac{\bar{M}_B}{2}$. The bounds in Theorem 2 are improved to $\left\lceil \log_{\frac{M_{\alpha q}+1}{2}} \frac{|C|}{\bar{M}_B} \right\rceil$.

From the theory of Monotonous Bisector* Tree we are able to transfer the following result:

Theorem 5: Under the assumptions of Theorem 2 the cluster radii on a path from the root to a leaf can be estimated by a geometrically decreasing sequence (geometric k–step development).

4. Queries

The C-tree provides the same queries as the Monotonous Bisector* Tree:

- nearest neighbor queries
- fixed–radius–near neighbor queries
- ray–shooting queries

- range queries
- points/objects in polygon retrieval
- objects hitting polygon retrieval
- objects hitting curve retrieval
- hidden–line/surface retrieval
- special problems in motion planning.

The way queries are performed is common to all rooted trees. Starting at the root all sons are explored recursively, if their subtrees are relevant for the query (*pruning of subtrees*). This can easily be checked examining the cluster of the sons. Furthermore exact matching leads to a unique search path, because of (O2). The geometrically decreasing radii guarantee an efficient execution of the queries. In contrast the Günther–Celltree only supports range and point queries.

When implementing a query, one has to remember that the position in the tree is determined by a CT–node $v_{\mathrm{CT}} \in V$ and a MBT–node $v_{\mathrm{MBT}} \in V(v_{\mathrm{CT}})$. To avoid page faults it is best to search an entire CT–node first, before recursively loading any successor page. Therefore the recursive calls must be stored in a stack.

Because the creation of the C-tree is sensitive in respect to the distance function d, also queries that depend on d (nearest neighbor, fixed–radius–near neighbor) are supported. Thereby a query object must be comparable to an object in S (cluster object). Let generally be Q a set of query objects and $\bar{d} : Q \times E \mapsto \mathbb{R}_{\geq 0}$, $\tilde{d} : Q \times S \mapsto \mathbb{R}_{\geq 0}$ the distance functions between query objects and split values resp. cluster objects. Then we additionally demand two triangle inequalities. For all $q \in Q$, $e \in E$, $s \in S$ holds:

$$\tilde{d}(q,s) \leq \bar{d}(q,e) + d(e,s) \tag{2}$$

$$\bar{d}(q,e) \leq \tilde{d}(q,s) + d(e,s) \tag{3}$$

In the case $Q = E = \mathbb{R}^N$, $S \subset \mathcal{P}(\mathbb{R}^N)$ the demanded properties reduce to the given distance function d. Inequality (2) allows to completely accept a subtree and Eq. (3) to prune a subtree.

First we consider the search for the fixed–radius–near neighbor of $q \in Q$ and the radius MAXDIST. That means we want to retrieve all $s \in S$ with $\tilde{d}(q,s) \leq$ MAXDIST. Let (S', e') be the cluster S' of the actually tested subtree with the cluster center e'. With $r(S', e')$ we denote the radius of the Cluster. Then it follows through Eq. (3): $\forall s \in S' : \tilde{d}(q,s) \geq \bar{d}(q,e') - d(e',s) \geq \bar{d}(q,e') - r(S',e')$. If we now have the situation that:

$$\bar{d}(q,e') - r(S',e') > \text{MAXDIST},$$

then $\tilde{d}(q,s) >$ MAXDIST for all $s \in S'$. So the entire subtree can be pruned.

For the search of the nearest neighbor of $q \in Q$ the following approach can be used: When already a good candidate $k \in S$ is derived (e.g. through searching the leaf with the nearest cluster center to q and taking any of the objects stored there), we can set DACTUAL $:= \tilde{d}(q,k)$. Then we can use the same arguments like before. A subtree can be pruned, if $\tilde{d}(q,e') - r(S',e') \geq$ DACTUAL, where e' denotes the cluster center and S' the represented objects of the actual node. Here we have $\geq$ instead of $>$, because objects with the same distance aren't better candidates.

Fig. 5 shows an example in $\mathbb{R}^2$ equipped with the Euclidean metric. The subtree related to the cluster (S_2, e_2) must be searched, whereas the subtree of cluster (S_1, e_1) can be pruned although e_1 is nearer to q than e_2 is. In fact for $l \in S_2$ there is $\tilde{d}(q,l) < \tilde{d}(q,k)$. If we find a better candidate than k, then we have to reset DACTUAL and continue the search.

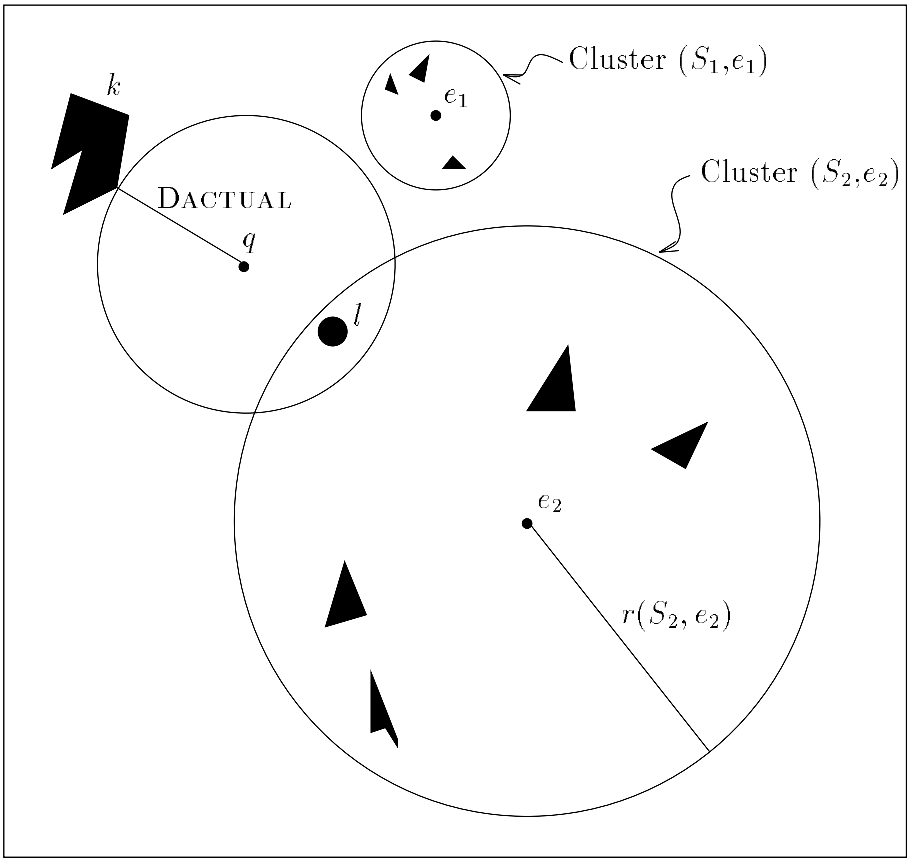

Fig. 5. Pruning of subtrees when searching the nearest neighbor.

5. Dynamic Operations: Inserting and Deleting Objects

The C-tree should support insertion and deletion of cells without degenerating, i.e. Theorem 2 and Theorem 4 are valid in the dynamic case with perhaps worse constants. To prove this we start with a definition:

Definition 6: An internal node where all sons are leaves, is called *B–node*. All other internal nodes are named *I–node*.

When we view a tree without its leaves, then B–nodes are exactly the new leaves and I–nodes the new internal nodes.

The derivation of Theorem 2 relies on the fact that for each I–node at least $M_{\alpha q}$ filled sons $V' \subset V$ exist for which Eq. (1) holds. If $\mathrm{Card}(v)$ denotes the cardinality of the objects in the subtree with root v, then:

$$\max_{v' \in V'} \mathrm{Card}(v') \leq 2 \min_{v' \in V'} \mathrm{Card}(v')$$

We now demand the weaker statement that after each operation still holds: For each I–node $v \in V$ there is a subset $V' \subset N(v)$, where $N(v)$ is the set of sons of v, with the following properties:

$$\begin{aligned} |V'| &\geq M_{\alpha q} \\ \min_{v' \in V'} \mathrm{Card}(v') &\geq \max_{v \in N(v) \setminus V'} \mathrm{Card}(v) \\ \max_{v' \in V'} \mathrm{Card}(v') &\leq 4 \min_{v' \in V'} \mathrm{Card}(v') \end{aligned} \tag{4}$$

Instead of the number 4 in the last inequality, we could have chosen any other real number > 2. The idea is to give each node a linear clearance, where it is still filled.

From Eq. (4) it follows, that each I–node has got at least a load of αq. This is all we want to demand, since the subtree of underfilled nodes may be deleted without notice. It is only important that always $M_{\alpha q}$ filled sons are present.

We start with a C-tree, which is constructed by Algorithm 1. So Eq. (4) is fulfilled. Now we must ensure, that Eq. (4) is still valid after each dynamic operation. This provides the logarithmic height and a minimum load on the average. To be able to test Eq. (4), we have to store $\mathrm{Card}(v)$ for each node v within the tree.

For rebalancing we use the static creation Algorithm 1, which fulfills Eq. (4). Since α is still undefined, we can set $\alpha := 1$.

Algorithm 2: Balance (v)

1. Store all objects in the subtree of v in $\mathrm{Set}(v)$ and delete the entire subtree except the root v itself.

2. Create (v, $\mathrm{Set}(v)$, e, α, B) where e is the split value of the MBT–father of v.

At this point we have the opportunity to enlarge the clearance of balancing, if we choose $\alpha < 1$. If the page v which caused the error is not full, we may only rebalance the subtree in $\mathrm{MBT}(v)$ instead of the entire CT–node. Using very large pages this reduces the rebalancing effort. But it does not affect the following worst–case results.

So we are now able to explain the *algorithms for inserting and deleting objects*. Because they are very similar, we will describe the insertion procedure. The variant of deleting objects is shown by the changes in brackets.

Algorithm 3: Insert [Delete] (w, o)

w : root of the C-tree

o : cell to be inserted [deleted]

1. (O2) provides the unique search path W to @Object(v_{MBT}) in v_{CT}.
   IF $o \in [\notin]$ @Object(v_{MBT}) is already present [not present], THEN STOP.

2. Insert [Delete] o in [from] @Object(v_{MBT}).
   (* this may cause a temporary overflow [underflow] *)

3. Update the cluster radii on the search path W.

4. Starting at the root check all I–nodes $v \in W \setminus \{v_{\mathrm{CT}}\}$:
   IF Eq. (4) is violated by v, THEN Balance(v) and STOP.

5. IF an overflow [underflow] of v_{CT} occurred, i.e. $|$@Set$(v_{\mathrm{MBT}})| > B$ [$= 0$], THEN:
   IF Father(v_{CT}) is a B–node,
   - THEN Balance(Father(v_{CT}))
   - ELSE Balance (v_{CT}) [IF v_{CT} is empty, THEN delete v_{CT}].

In the undesired case, that the object determining the cluster radius of $v \in V$ is deleted, the new farthest object must be found. Therefore we may use the given spatial index. But in the worst–case the time effort is still $\mathcal{O}(\mathrm{Card}(v))$ for each node on the search path. This summarizes to $\mathcal{O}(|C|)$ for updating the cluster radii on the search path. Another possibility is to compute the cluster radii delayed in the background. Thus after deletion they may be temporarily too large. But the effort in the amortized case would then be equal to the insertion effort.

Step 3 of the algorithm shows, that (O1) is fulfilled. From 4. it follows Eq. (4) and from step 5 (P). So we have:

Theorem 6: Let $M_{\alpha q} := \lfloor (\lceil M\alpha \rceil + 1) q \rfloor$, $q := \frac{1}{3}$, $\alpha := 1$, the cells be convex polyhedrons in $\mathbb{R}^N$ and d any L_p–metric. After an arbitrary sequence of insertions and deletions the C-tree has got a height of at most

$$\begin{cases} \left\lceil \log_{\frac{M_{\alpha q}+3}{4}} \frac{|C|}{B} \right\rceil, & \text{if } |C| > B \\ 0, & \text{if } |C| \leq B \end{cases},$$

if at most each second partition step is not balanced and $M_{\alpha q} \geq 5$.

Theorem 7: Under the assumptions of Theorem 6 after an arbitrary sequence of insertions and deletions the C-tree has got $\mathcal{O}(|C|)$ space requirement in secondary memory. More precisely the following bounds hold: For all internal nodes the load is $\geq \alpha q$, up to one exception on a path from the root to a leaf.

Since inserting or deleting an object may cause a complete creation of the tree, we have:

Theorem 8: The effort to insert or delete in the C-tree is $\mathcal{O}(|C| \log |C|)$ time in the worst–case.

We will now examine insertion and deletion more precisely, in order to obtain a statement about the *amortized costs*. In Algorithm 3 the search path in step 1, 3 and in test Eq. (4) in step 4 is passed once in $\mathcal{O}(\log |S|)$. When deleting we update the cluster radii immediately in $\mathcal{O}(|S|)$ or delayed in $\mathcal{O}(\log |S|)$. The steps 2 and 5 can be executed in $\mathcal{O}(1)$, since the page size is fixed.

Altogether the total time to insert or delete with immediate (resp. delayed) updating of the cluster radii is logarithmic (resp. linear), if in step 4 no rebalancing is necessary. The following theorem now shows how expensive this rebalancing is.

Theorem 9:

1. The amortized time effort for inserting in the C-tree is $\mathcal{O}(\log^2 |C|)$.
2. The amortized time effort for deleting in the C-tree is $\mathcal{O}(\log^2 |C|)$, if updating of cluster radii is delayed.
3. The amortized time effort for deleting in the C-tree is $\mathcal{O}(|C|)$, if cluster radii are updated immediately.

Proof: It suffices to examine the additional costs for rebalancing in step 4. Therefore we only have to consider the I–nodes.

Let $v_{\mathrm{CT}} \in V$ be a I–node. How many insertion and deletion operations $o(v_{\mathrm{CT}})$ are at least necessary to force v_{CT} to be rebalanced? After executing Balance(v_{CT}) it is according to Eq. (1) for the $M_{\alpha q}$ largest sons V' of v_{CT}:

$$\max_{v' \in V'} \mathrm{Card}(v') \leq 2 \min_{v' \in V'} \mathrm{Card}(v').$$

If we now have to rebalance v_{CT} for the first time *after an arbitrary sequence of e insertion and l deletion operations* (i.e. $o(v_{\mathrm{CT}}) = e + l$), then Eq. (4) is violated. For the worst–case (inserting in the largest and deleting in the smallest cluster) there is:

$$\max_{v' \in V'} \mathrm{Card}(v') + e \quad \stackrel{(4)}{>} \quad 4(\min_{v' \in V'} \mathrm{Card}(v') - l) \quad \stackrel{(1)}{\geq} \quad \max_{v' \in V'} \mathrm{Card}(v') + 2 \min_{v' \in V'} \mathrm{Card}(v') - 4l$$

$$\Rightarrow \quad 4o(v_{\mathrm{CT}}) \quad \geq \quad e + 4l \quad > \quad 2 \min_{v' \in V'} \mathrm{Card}(v') \quad \geq \quad \frac{2\mathrm{Card}(v_{\mathrm{CT}})}{\alpha M}.$$

This means after at least

$$o(v_{\mathrm{CT}}) = \frac{\mathrm{Card}(v_{\mathrm{CT}})}{2\alpha M} \tag{5}$$

operations a rebalancing of v_{CT} with effort $A_{v_{\mathrm{CT}}} = \mathcal{O}(\mathrm{Card}(v_{\mathrm{CT}}) \log \mathrm{Card}(v_{\mathrm{CT}}))$ may occur. The amortized effort for this is therefore $\mathcal{O}(\log \mathrm{Card}(v_{\mathrm{CT}}))$.

We now consider the *total time for a worst–case sequence of length $o(v_{\mathrm{CT}})$ which forces v_{CT} to be rebalanced*. Let $N := \mathrm{Card}(v_{\mathrm{CT}})$. How large is the rebalancing effort for the sons of v_{CT}? Let v_i ($i = 1, \ldots, m$) be the son of v_{CT}, which is affected when the ith rebalancing of a son of v_{CT} occurs. Let $N_i := \mathrm{Card}(v_i)$. Then the total time for all sons of v_{CT} is:

$$A_S = \sum_{i=1}^{m} \mathcal{O}(N_i \log N_i)$$

The guaranteed reduction factor in the dynamic case is $K := \dfrac{M_{\alpha q} + 3}{4}$. Thus for all $i = 1, \ldots, m$ it is: $N_i \leq \frac{N}{K}$. Since $v_1, \ldots, v_m$ were rebalanced, we have:

$$\sum_{i=1}^{m} N_i \overset{(5)}{=} 2\alpha M \sum_{i=1}^{m} o(v_i) \leq 2\alpha M o(v_{\mathrm{CT}}) \overset{(5)}{=} N$$

and therefore

$$A_S \leq \sum_{i=1}^{m} \mathcal{O}\left(N_i \log \frac{N}{K}\right) \leq \mathcal{O}\left(N \log \frac{N}{K}\right).$$

So the total cost for the subtree of v_{CT} with height h is determined by:

$$\begin{aligned} A &= A_{v_{\mathrm{CT}}} + A_S + \ldots \\ &\leq \mathcal{O}\left(N \log N + N \log \frac{N}{K} + N \log \frac{N}{K^2} + \ldots + N \log \frac{N}{K^h}\right) \\ &= \mathcal{O}\left(N \left(\log N + \log \frac{N}{K} + \log \frac{N}{K^2} + \ldots + \log \frac{N}{K^h}\right)\right) \\ &\leq \mathcal{O}\left(N \left(\log N + h \log \frac{N}{K}\right)\right) \\ &\leq \mathcal{O}\left(N \left(\log N + \log \frac{N}{B} \log \frac{N}{K}\right)\right) \end{aligned}$$

We conclude that the amortized effort for balancing is $\mathcal{O}(\log^2 N)$.

Concerning item 1 and 2 of the theorem we have proved that the time effort for deletion with delayed update of cluster radii and for insertion is (inserting + rebalancing):

$$\mathcal{O}(\log N) + \mathcal{O}(\log^2 N) = \mathcal{O}(\log^2 N).$$

Concerning item 3 we showed that the time effort for deletion with immediate updating of cluster radii is:

$$\mathcal{O}(N) + \mathcal{O}(\log^2 N) = \mathcal{O}(N).$$

□

6. Concluding Remarks

The C-tree has been implemented[14] with different scenes and queries. It was tested extensively and compared to the Monotonous Bisector* Tree. These tests show that the Monotonous Bisector* Tree and C-tree are flexible and robust indices for managing geometric spatial data. Nevertheless both data structures are not competing, but supporting each other in different aims and applications. Both data structures are applicable for different types of geometric queries, where we have observed optimal query times in tests.

The Monotonous Bisector* Tree is especially applicable in small scenes ($\mathcal{O}($ main memory $)$), because it is efficient and easy to implement. When managing very complex objects the Monotonous Bisector* Tree is advantageous, too.

But in very large scenes the C-tree is clearly superior. The field of applications is only restricted through the available secondary memory. Furthermore the C-tree is distinguished through its very *cooperative* character. In today's multi–tasking environments it does not compete with other processes, because it requires only a constant amount of main memory.

In our future research we will explore the application of spatial indices in motion planning[1,5,15]. Since the time complexity of motion planning depends of the complexity of the scene, we want to use spatial indices to reduce the input complexity. First, we may use the index as a filter for locally relevant data. Second, we can combine the retrieved clustering with incremental search methods. And third, we may group close objects and approximate them by a container. This grouping must be refined, if no adequate path can be found.

References

1. C. Ballieux. Motion Planning using Binary Space Partition. Technical Report inf/src/93-25, Utrecht University, 1993.
2. R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1:173–189, 1972.
3. J.L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.
4. H. Fuchs, Z. Kedem, and B. Naylor. On Visible Surface Generation by a Priority Tree Structure. *Computer Graphics*, 14(3):124–133, 1980.
5. K. Fujimura and H. Samet. Path Planning among Moving Obstacles using Spatial Indexing. In *IEEE International Conference on Robotics and Automation*, pages 1662–1667, 1988.
6. O. Günther. *Efficient Structures for Geometric Data Management*, volume 337 of *Lecture Notes in Computer Science*. Springer, 1988.
7. O. Günther and J. Bilmes. The Implementation of the Cell Tree: Design Alternatives and Performance Evaluation. *Informatik–Fachberichte*, 204:246–265, 1989.
8. O. Günther and H. Noltemeier. Spatial Database Indices for Large Extended

Objects. In *Proceedings IEEE — 7th International Conference on Data Engineering*, Kobe (Japan), April 1991.
9. A. Guttmann. R–Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD*, pages 47–57, 1984.
10. J. Nievergelt. 7 ± 2 Criteria for Assessing and Comparing Spatial Data Structures. *Lecture Notes in Computer Science*, 409:3–27, 1990.
11. H. Noltemeier, K. Verbarg, and C. Zirkelbach. A Data Structure for Representing and Efficient Querying Large Scenes of Geometric Objects: MB^* Trees. In G. Farin, H. Hagen, and H. Noltemeier, editors, *Geometric Modelling*, pages 211–226. Springer, 1993.
12. B.C. Ooi. *Efficient Query Processing in Geographic Information Systems*, volume 471 of *Lecture Notes in Computer Science*. Springer, 1990.
13. F.P. Preparata and M.I. Shamos. *Computational Geometry — An Introduction*. Springer, 1985.
14. K. Verbarg. Räumliche Indizes — Celltrees: Analyse und experimenteller Vergleich mit Monotonen Bisektorbäumen. Master's thesis, Universität Würzburg, 1992.
15. D. Zhu and J.-C. Latombe. New Heuristic Algorithms for Efficient Hierarchical Path Planning. *IEEE Transactions on Robotics and Automation*, 7(1):9–20, 1991.
16. C. Zirkelbach. *Geometrisches Clustern — ein metrischer Ansatz*. PhD thesis, Universität Würzburg, 1992.