O-Trees: a Constraint-based Index Structure

Inga Sitzmann and Peter Stuckey
Department of Computer Science and Software Engineering
The University of Melbourne
Parkville, Victoria, 3052
{inga,pjs}@cs.mu.oz.au

Abstract

Constraint search trees are a generic approach to search trees where all operations are defined in terms of constraints. This abstract viewpoint makes clear the fundamental operations of search trees and immediately points to new possibilities for search trees. In this paper we present height-balanced constraint search trees (HCSTs), a general approach to building height-balanced index structures, and exemplify the approach with a new spatial index structure, the O-tree. An object in an O-tree is represented by constraints of the form $ax_i + bx_j \leq d$ *where* $\{a, b\} \subseteq \{-1, 0, 1\}$ *and* $x_1, \ldots, x_n$ *are the dimensions of the spatial data. We define the basic operations to build and search HCSTs, as well as constraint joins. We illustrate these algorithms using O-trees showing how the algorithms can make use of the more accurate information in the O-tree nodes. Experiments compare the IO-performance of the 2-dimensional O-tree with the R-tree.*

1. Introduction

Search trees are a fundamental data structure of computer science, providing a way of storing collections of objects which allows efficient access, insertion and deletion by key value. Many variants of search trees have been defined and studied such as binary search trees, radix search trees, k-d-trees [1], B-trees and R-trees [2]. Constraint search trees (CSTs) defined in [8] abstract the fundamental behaviour of a search tree in terms of constraints.

CSTs store data items in the form of constraints (in practice a constraint key is used to store arbitrary items), and constraints are used to control the search in the tree. CSTs were originally defined as binary trees. In this paper we define height-balanced variations of constraint search trees, and exemplify them by defining a new spatial index structure, the O-tree.

The rapid increase of available information on data used in fields such as Geography, Cartography and Earth Sciences has lead to demand for efficient systems managing the underlying *spatial data*. Spatial data refers to all kinds of geometric objects, such as arcs and polygons, and their location and extent in space. Important applications of spatial data are Geographic Information Systems (GIS) which allow data entry, data display, and data management of spatial information. The technology behind these systems are spatial databases which store spatial data and allow efficient access to the data stored. Several index structures for spatial databases have been proposed including commonly used ones such as R-trees [2] and quad trees [7]. Furthermore, join algorithms have been developed to provide efficient evaluation of spatial join queries. All data structures and join algorithms are based on approximating spatial objects by objects of simpler geometric shape, usually circles or rectangles. Bad approximations can result in a high number of unnecessary block accesses and therefore in a bad overall performance of the index structure. Adding more information about objects, on the other hand, leads to more complicated operations on the index as well as a higher demand of storage space. In this paper we present a new spatial data structure, the O-Tree, in the general framework of height-balanced constraint search trees, and investigate whether its extra information can improve IO performance.

2. Constraint Search Trees

Constraint search trees are a general framework for search trees based on the notions of constraint satisfaction and entailment [8]. In this paper we introduce height-balanced constraint search trees and exemplify their use with O-trees. First we quickly review the original definitions.

2.1. Binary Constraint Search Trees

In general, search trees consist of a set of external or leaf nodes, and a set of internal or directory nodes. Leaf nodes contain the data stored in the tree, whereas directory nodes usually consist of directory information, i.e. discriminators, that lead search through the tree to the node where an entry can be found. In the constraint search tree view, both

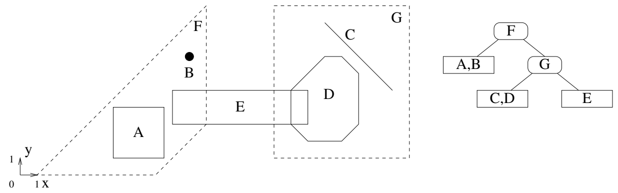

Figure 1. A CST for octagon objects

data items (or keys for data items) and discriminators are represented by constraints and constraint entailment is the mechanism used to direct search.

Constraint search trees as defined in [8] are binary trees consisting of two types of nodes:

- external nodes of the form *ext(C)* containing a set of constraints C (each constraint is a data item); and
- internal nodes of the form *int(d,t,u)* where d is a discriminator constraint, t and u are constraint trees, such that each data item c occurring in tree t implies the discriminator d (i.e. $c \to d$).

The formal definition of constraint trees is given by specifying a constraint domain $\mathcal{A}$. A constraint domain $\mathcal{A}$ consists of: a signature defining the language of constraints (the function and relation symbols), and an interpretation $\mathcal{M}_\mathcal{A}$ defining the meaning of each function and relation symbol in the signature, together with a set of first order formulae $\mathcal{F}_\mathcal{A}$ defining the acceptable constraints of the domain. We assume that $\mathcal{F}_\mathcal{A}$ is closed under conjunction. In practice, we will restrict the possible discriminator constraints to a set $\mathcal{D}_\mathcal{A} \subseteq \mathcal{F}_\mathcal{A}$.

In practice, to use a constraint search tree for storing and accessing constraints from a domain $\mathcal{A}$, two functions have to be provided. A function $\mathsf{satisfiable} :: \mathcal{F}_\mathcal{A} \to \{true, false, unknown\}$ which determines whether a constraint is satisfiable or not, and a function $\mathsf{implies} :: \mathcal{F}_\mathcal{A} \times \mathcal{F}_\mathcal{A} \to \{true, false, unknown\}$ which determines whether one constraint implies another.

The functions satisfiable and implies are meant to reflect the meaning of constraints given by the interpretation of $\mathcal{A}$. They may be *incomplete*, that is sometimes return *unknown*. Irrespective of their completeness property, the functions must satisfy the following correctness conditions:

$$\begin{array}{rcl} \text{if } \mathsf{satisfiable}(c) & = & true \text{ then } \mathcal{M}_\mathcal{A} \models \exists c \\ & = & false \text{ then } \mathcal{M}_\mathcal{A} \models \neg\exists c \\ \text{if } \mathsf{implies}(c_1, c_2) & = & true \text{ then } \mathcal{M}_\mathcal{A} \models \forall c_1 \to c_2 \\ & = & false \text{ then } \mathcal{M}_\mathcal{A} \models \neg\forall(c_1 \to c_2) \end{array}$$

Example 1 Consider a constraint domain $\mathcal{O}$ of conjunctions of constraints of the form $ax + by \leq c$, where $\{a, b\} \subseteq \{0, 1, -1\}$, $c \in \Re$ and variables x and y range over real numbers. These define convex two dimensional areas ((x, y) points) that are bounded by lines at angles $0, \pi/4, \pi/2$ and $3\pi/4$. These areas can be at most octagonal in complexity.

Consider the objects A to G on the left of Figure 1. Each can be represented by a constraint in domain $\mathcal{O}$. For example, F is $y \geq 0 \wedge x \leq 11 \wedge y + 1 \leq x \wedge y + 8 \geq x$ and B is $x \geq 10 \wedge x \leq 10 \wedge y \geq 7 \wedge y \leq 7$. Note how containment corresponds to entailment, e.g. $B \to F$.

A CST storing the items A, B, C, D and E is represented at the right of Figure 1, where F and G are discriminator constraints. It corresponds to the CST $int(F, ext(\{A, B\}), int(G, ext(\{C, D\}), ext(\{E\})))$. $\square$

Algorithms for constraint search trees are usually straightforward and make use of the algorithms satisfiable and implies. For example, the pseudo-code in Figure 2 finds all the constraints in a tree T which intersect with a query constraint c.

```
intersect(T, c)
  case T of
  ext(C): return {c' | c' ∈ C, satisfiable(c ∧ c') ≠ false}
  int(d, t, u): A := {}
    if (satisfiable(c ∧ d) ≠ false)
        A := A ∪ intersect(t, c ∧ d)
    return A ∪ intersect(u, c)
```

Figure 2. Intersection search in a CST

Importantly, as the tree is traversed the discriminator constraints which are known to hold are collected in order to narrow further search. This is not required for correctness of the algorithm but can be of crucial importance in reducing the search required, because satisfiable and implies are possibly incomplete.

Example 2 Consider a constraint domain $\mathcal{O}^+$ for conjunctions of linear real constraints where discriminators are restricted to be $\mathcal{O}$ constraints ($D_{\mathcal{O}^+} = F_\mathcal{O}$). An incomplete constraint solver for $\mathcal{O}^+$ treats the more complicated constraints (not in $\mathcal{O}$) as propagators (see e.g. [6]) to generate new $\mathcal{O}$ information. For example, the non $\mathcal{O}$ constraint $x + 2y \leq 5$ together with $\mathcal{O}$ inequality $y \geq 1$ generates the following implied $\mathcal{O}$ inequalities: $x \leq 3$, $x + y \leq 4$ and $x - y \leq 2$. This solver can be used for searching trees with $\mathcal{O}$ discriminators. For example, consider searching for items intersecting the line shown in Figure 3, the solver maintains a bounding $\mathcal{O}$ constraint (shown by the dotted line) for the query line. When the query constraint is conjoined with the $\mathcal{O}$ discriminators $d1$ and $d2$, the solver determines new $\mathcal{O}$ bounding constraints $b1$ and $b2$ shown by the dashed lines through propagation. This means unsatisfiability of constraints can be detected more often. For example, while the original bounding constraint of the line intersects with the shaded region, $b2$ does not, hence an intersection search using the propagation solver will not ex-

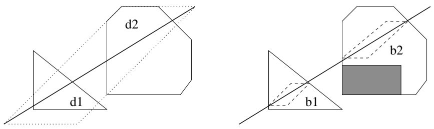

Figure 3. An incomplete constraint solver making use of extra constraint information

plore the subtree with a discriminator describing the shaded region. □

Note that in all algorithms for constraint search trees, if the algorithms satisfiable and implies are incomplete the result may be a superset of the actual (logical) answer. So for example intersect(T, c) may return a constraint c' in T which does not actually intersect with c, but this is not possible to know since satisfiable$(c \wedge c')$ returns $unknown$.

3. Height-balanced Constraint Search Trees

In the context of storing large amounts of data binary trees are not suitable, and many tree based storage methods rely on height-balanced trees with nodes that have many children, for example B-trees and R-trees. In this section we define a height-balanced version of constraint search trees which provides a uniform approach to storing constraint data in a height-balanced data structure.

A *height-balanced constraint search tree* (HCST) is made up of two kinds of nodes:

- external nodes $ext([c_1, \ldots, c_n])$ contain a sequence of data items (constraints).
- internal nodes $int([d_1, \ldots, d_n], [t_1, \ldots, t_n])$ where $[d_1, \ldots, d_n]$ is a sequence of discriminators and $[t_1, \ldots, t_n]$ is a sequence of child trees (in reality addresses of child trees).

The following conditions must hold for the tree to be a height-balanced constraint search tree:

- The root node has at least two children unless it is an external node.
- Each non-root node uses at least $1/k$ of its available space to store the objects (data items or discriminators and child pointers) in it, for some fixed k. For example, in an external node assuming each item requires the same space (certainly not always true in the constraint framework) and each node can contain M items, then each external node should contain at least $\lceil M/k \rceil$ entries.
- Every external node is the same distance from the root.
- Each data item c appearing in a child tree t_i of node $int([d_1, \ldots, d_n], [t_1, \ldots, t_n])$ is such that implies$(c, d_i) = true$.

Example 3 A height-balanced CST storing the items A, B, C, D and E shown in Figure 1 is

$$int([E, F, G], [ext([E]), ext([A, B]), ext([C, D])]).$$

Note the differences from the CST. The tree is not binary, and each external node requires a discriminator entry (so for example E is used as the discriminator for the external node containing only E). □

Intersection queries in HCSTs are completely analogous to the intersect code given above for CSTs. We can also similarly define *contains* (resp. *surround*) queries which search the tree to find constraints that possibly imply (resp. are implied by) the query constraint. In a geometric interpretation, they are contained by (resp. contain) the query. For example:

```
contains(T, c)
  case T of
  ext([c_1, ..., c_n]):
    return {c_i | 1 ≤ i ≤ n, implies(c_i, c) ≠ false}
  int([d_1, ..., d_n], [t_1, ..., t_n]): A := {}
    for i = 1 to n
      if (satisfiable(d_i ∧ c) ≠ false)
        A := A ∪ contains(t_i, c ∧ d)
    return A
```

3.1. Building Height-Balanced CSTs

Height-balanced CSTs are built by repeatedly inserting entries in the tree. Hence, inserting an entry has to maintain a height-balanced tree. This is achieved by a splitting function which distributes a set of entries into two nodes and propagates the resulting information upwards. The quality of the splitting algorithm highly influences the performance of further operations, e.g. searches, in the tree. A good split of a node distinguishes as much as possible between the entries, thus leading to a more efficient search.

In building height-balanced trees we will need to introduce a number of functions specific to the constraint domain $\mathcal{A}$ in order to create correct HCSTs after insertion of a new data item. union$([c_1, \ldots, c_m])$ returns a discriminator d which is implied by each of $c_1, \ldots, c_m$, that is implies$(c_i, d) = true$, and in addition for any c if implies$(c, c_i) = true$ and $d =$ union$([c_1, \ldots, c_m])$ then implies$(c, d) = true$. fits(t) determines if a node t fits within the space allocated to store nodes of that type. If not the node has to be split. measure(c) returns a numeric measure of the size of constraint c in terms of number of solutions, or selectivity. The smaller the size of the constraint, the more preferable it is for use as a discriminator.

An HCST is built by repeated insertion using insert(T, e) shown in Figure 4. The entry e is inserted in the tree T using ins. This procedure always returns a tree t' with topmost int node containing one or two subtrees. If

```
insert(T, e)
    t' := ins(T, e)
    if t' = int([d], [t]) return t
    return t'

ins(T, e)
    case T of
    ext([d_1, ..., d_n]):
        t := ext([d_1, ..., d_n, e])
        if fits(t) return int([union([d_1, ..., ..., d_n, e])], [t])
        else
            (D_1, D_2) := split([d_1, ..., d_n, e])
            d_1 := union(D_1); d_2 := union(D_2);
            return int([d_1, d_2], [ext(D_1), ext(D_2)])
    int([d_1, ..., d_n], [t_1, ..., t_n]):
        i := choose_subtree([d_1, ..., d_n],e)
        t' := ins(t_i, e)
        let t' = int(F, S)
        t := int([d_1, ..., d_{i-1}]++F++[d_{i+1}, ..., d_n],
                [t_1, ..., t_{i-1}]++S++[t_{i+1}, ..., t_n])
        d := union([d_1, ..., d_{i-1}]++F++[d_{i+1}, ..., d_n])
        if fits(t) return int([d], [t])
        else
            let t = int([c_1, ..., c_m], [r_1, ..., r_m])
            (C_1, C_2) := split([c_1, ..., c_m])
            d_1 := union(C_1); R_1 := corresponding trees to C_1
            d_2 := union(C_2); R_2 := corresponding trees to C_2
            return int([d_1, d_2], [int(C_1, R_1), int(C_2, R_2)])
```

Figure 4. Insertion into an HCST

it only contains a single subtree t, then t is the result of the insert, otherwise the result is t'.

The entry e to be inserted is recursively handed down to the external node where it has to be added. In an external node, the new constraint e is added to the node if this is possible without overflowing the node. Otherwise split is called to split the set of entries in the node. In an internal node, the procedure is the same. If after inserting e in the subtree t_i the resulting internal node fits within the space available the new subtrees are just used to replace t_i. If this would overflow the node the splitting algorithm is called and two nodes produced.

The procedure choose_subtree picks the entry in a node whose discriminator needs the least enlargement to include the new item, returning its index in the sequence.

```
choose_subtree([d_1, ..., d_n], c)
    min := ∞
    for i := 1 to n
        m := measure(union([c, d_i])) − measure(d_i)
        if m < min then min := m; mi := i
    return mi
```

The split algorithm is also another requirement on the constraint domain. It splits a sequence of constraints into two sequences which will each fit in a node and are of some minimum size ($1/k$th of a node). It should try to minimize the overlap and measure of the unions of the resulting sequences. One can define splitting algorithms just using the already introduced functions, for example by simply trying every possible split (of appropriate sizes) and picking the split with minimal total of the two measures plus the measure of their intersection.

Deletion can be defined similarly to insertion.

```
sameheight_join(T_1, T_2, c)
    case T_1 of
    ext([d_1, ..., d_n]):
        let T_2 = ext([f_1, ..., f_m])
    int([d_1, ..., d_n], [t_1, ..., t_n]):
        let T_2 = int([f_1, ..., f_m], [s_1, ..., s_m])
    endcase
    J := {}; R := {}
    for j = 1 to m
        if (satisfiable(c ∧ f_j) ≠ false) J := J ∪ {j}
    for i = 1 to n
        if (satisfiable(d_i ∧ c) ≠ false)
            for j ∈ J
                if (satisfiable(d_i ∧ f_j ∧ c) ≠ false)
                    if (T_1 = ext(_)) R := R ∪ {(d_i, f_j)}
                    else
                        R := R ∪
                        sameheight_join(t_i, s_j, c ∧ d_i ∧ f_j)
    return R
```

Figure 5. The constraint join for HCSTs of the same height

3.2. Joining HCSTs

An important operation for index structures to support is an efficient join. We consider the *constraint join* $R_1 \bowtie_c R_2$ where relations R_1 and R_2 are represented by HCSTs T_1 and T_2 respectively, and the result of the join is the set $\{c_1 \wedge c \wedge c_2 \mid c_1 \in R_1, c_2 \in R_2\}$.

We give the algorithm for joining HCSTs T_1 and T_2 of the same height in Figure 5.

The steps are almost identical for internal or external nodes. If T_1 is an external node, then T_2 is as well, similarly for internal nodes. The set of potential join partners in each node is reduced to those which are (possibly) satisfiable when conjoined with the parent constraint. Only entries which meet this condition can be potential join partners. The next step is to check the satisfiability of each remaining discriminator/item conjoined with each discriminator/item of the other set and the parent constraint. If the nodes are external the pair of items is added to the result set, otherwise the final step involves a recursive call of the join algorithm for each pair of subtrees whose discriminators are (possibly) satisfiable when conjoined. This call conjoins the

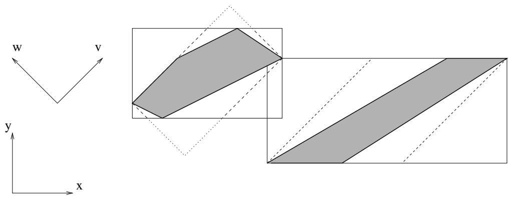

Figure 6. Approximation of polygons in a R-tree and O-tree

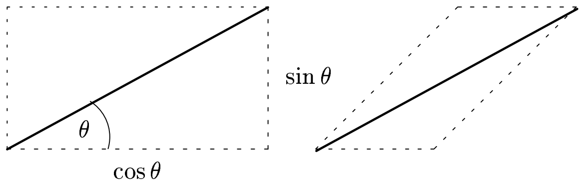

Figure 7. Size of the O-tree bounding box and the R-tree bounding box for line data

conjunction of the discriminators of the child trees to the constraint of the join. Since each element in the corresponding trees implies these constraints, this does not change the solutions, but it does make information available earlier to the constraint solver which may help reduce the amount of work required for the join.

Joining trees of different height is nearly the same as joining trees of the same height until the external level is reached in one of the trees. Then, a second algorithm is necessary to find join partners for each entry in the external node in the other tree, searching the remaining subtrees.

4. O-trees

This section defines O-trees in the general framework of height-balanced constraint search trees. We shall concentrate on O-trees for 2-dimensional objects, the approach extends naturally to 3 or more dimensions. Many search tree index structures, e.g. R-trees, approximate 2-d data by using a minimum bounding box (mbb), thus the mbb represents the key for the data item. As shown in Figure 6, for some kinds of data this is a very poor representation. Although, the two shaded polygons are far from intersecting each other, an intersection test based on the mbbs indicates an overlap.

We can consider an O-tree as an example of a constraint search tree where the discriminator constraints are conjunctions of inequalities of the form $ax_i + bx_j \leq d$ where $x_1, \ldots, x_n$ are the n dimensions of objects to be stored, $i \neq j$, and $\{a, b\} \subseteq \{0, 1, -1\}$. These constraints are known as unit two-variable per inequality (UTVPI) constraints [4]. In 2 dimensions these are exactly the constraints from domain $\mathcal{O}$. The $\mathcal{O}$ constraint keys for the polygons of Figure 6 are also shown (dashed), and here the lack of overlap is clear.

One can think of the additional complexity of an $\mathcal{O}$ discriminator as representing another bounding box along the two additional axes v and w (see Figure 6), which leave the origin at an angle of $\pi/4$ to the x and y axes (more precisely $v = (x + y)/\sqrt{2}$ and $w = (y - x)/\sqrt{2}$). Any constraint from domain $\mathcal{O}$ defining a bounded region is representable in the following form:

$$xl \leq x \wedge x \leq xu \wedge yl \leq y \wedge y \leq yu \wedge$$

$$v = (x + y)/\sqrt{2} \wedge w = (y - x)/\sqrt{2} \wedge$$
$$vl \leq v \wedge v \leq vu \wedge wl \leq w \wedge w \leq wu.$$

Thus an $\mathcal{O}$ constraint can be represented by the 8 constants $xl, xu, yl, yu, vl, vu, wl, wu$ occurring in the constraint description. O-trees are particularly appropriate for storing line data. When storing a 2-d unit length line at an angle $0 \leq \theta \leq \pi/8$ to the horizontal, the area of the bounding box is $\cos(\theta)\sin(\theta)$. In an O-tree, on the other hand, the area of the intersection of the bounding boxes is $\cos(\theta)\sin(\theta) - \sin^2(\theta)$. This means that the O-tree region bounding a line is on average $2 - \frac{\pi}{2} \approx 0.43$ times the area of the R-tree minimum bounding box. This is illustrated in Figure 7. The benefit improves further in higher dimensions.

4.1. O-trees as an example of the HCST framework

To use O-trees in the HCST framework we need to specify the various required algorithms. We restrict the discussion to 2-d O-trees for simplicity. The basic algorithms satisfiable and implies are straightforward to define. Using an 8-tuple representation of a 2-d O-tree constraint as $(xl, xu, yl, yu, vl, vu, wl, wu)$ we can define a normal form algorithm which tightens the bounds of the x and y with respect to those of v and w and vice versa. The conjunction of two constraints is given by keeping the tightest bound of the two constraints in each of the 8 directions, and recomputing the normal form. Satisfiability is simply a matter of examining the normal form of the constraint and detecting if any variable has no possible values. Similarly, implication simply checks whether the normal form of the conjunction of the two constraints is identical to the normal form of the first argument, in which case every bound of the first argument is at least as tight as for the second argument.

The union$([d_1, \ldots, d_n])$ algorithm is defined by taking the convex hull of the discriminators $d_1, \ldots, d_n$, that is keeping the loosest bound in each direction. Appropriate measure functions for O-tree constraints are either measuring the area of the discriminator, or the length of the perimeter of the discriminator.

The splitting algorithm we use for O-trees is based on the splitting algorithm for R-trees defined by Guttman [2]. We pick two seed constraints from the sequence C which

Lines	R-Tree						O-Tree					
	Ht.	Index Acc.	Results	Leaves (hits)	Index + Results	Index + Leaves	Ht.	Index Acc.	Results	Leaves (hits)	Index + Results	Index + Leaves
100	2	287	100	100	387	387	2	264	100	100	364	364
500	3	3408	503	502	3911	3910	3	3302	501	501	3803	3803
1000	3	8970	1018	1014	9988	9984	4	9000	1012	1006	10012	10006
5000	4	82113	5455	5295	87568	87408	4	75047	5282	5197	80329	80244
10000	4	227514	12021	11517	239535	239031	5	208723	11312	10988	220035	219711
20000	4	612176	28427	26458	640603	638634	5	576152	25329	24174	601481	600326

Poly gons	R-Tree						O-Tree					
	Ht.	Index Acc.	Results	Leaves (hits)	Index + Results	Index + Leaves	Ht.	Index Acc.	Results	Leaves (hits)	Index + Results	Index + Leaves
100	2	768	4461	659	5229	1427	2	1128	3587	953	4715	2081
500	3	16985	95154	14565	112139	31550	3	28103	74190	21623	102293	49726
1000	3	67048	391216	58290	458264	125338	4	109686	303663	84747	413349	194433
5000	4	1621745	9578004	1423313	11199749	3045058	4	2631274	7433136	2048171	10064410	4679445

Table 1. R-trees versus O-trees for intersection queries on line (a) and polygon (b) data

maximize the normalized separation along one dimension x or y. The normalized separation in the x dimension is $\max_{d \in C}\{d.xl\} - \min_{d \in C}\{d.xu\} / \Sigma_{d \in C}(d.xu - d.xl)$. Assuming x is the dimension for separation, the two seeds are the constraint with minimum xu and the constraint with maximum xl. These form the starting point for two sets. The next step scans the remaining constraints adding each to the set which suffers the least increase in size of its convex hull. This splitting is linear in complexity as opposed to the exponential complexity of testing all possible splits. We experimented by also considering maximum separation on dimension v and w as well, but this splitting rule always gave worse trees than restricting to two orthogonal directions x and y.

5. Experimental results

We assess the quality of O-trees as an index structure for 2-d spatial data, by comparing them with R-trees. For a fair comparison, both tree structures are implemented as instances of the HCST framework (R-trees are also HCSTs with constraints $xl \leq x \wedge x \leq xu \wedge yl \leq y \wedge y \leq yu$) and use the same generic code for HCST operations.

The performance of O-trees is evaluated in two sets of experiments. The first set compares the performance of O-trees to R-trees in intersect queries, while the second set of experiments compares the joins.

The test data used are a set of randomly constructed line and polygon data relations. Each line data set contains a number of lines each with approximate length 20 in a square of area 5000. The polygon data sets contain convex polygons with up to 10 nodes and edges of length approximately 40 in a square area of 10000. The polygons are constructed by randomly creating the 10 points and using the Graham scan algorithm to calculate their convex hull. The key to each item is the smallest $\mathcal{O}$ constraint containing the line or polygon.

In our implementation, an O-tree entry, that is $\mathcal{O}$ constraint plus pointer to child tree (or pointer to actual data item for leaf nodes) requires $8 \times 32 + 64 = 320$ bits. Similarly, an R-tree entry requires $4 \times 32 + 64 = 192$ bits. We compare R-trees and O-tree that make use of a constant node size of 6400 bits, hence an O-tree node can store at most 12 entries, while an R-tree node stores at most 20 entries.

We measure efficiency of the methods in terms of the number of block accesses required to search the index and access the data. A block access takes place when a node has to be read from the disk to main memory. We give the number of index block accesses, plus two measures of the accesses required to read the data. The total number of block accesses depends on the clustering of the data. In the worst case, one block access is necessary to retrieve the data for each result. The best case is that data items for a leaf node all reside in one data block and only one data block access is required for each leaf node which contains an answer. The total number of block accesses will vary in between these two extremes depending on the clustering of data storage.

5.1. Intersection query comparison

In the intersection experiments we used bounds propagation solvers for both O-tree and R-tree queries. The constraint solvers use the constraints of the query to maintain a smallest bounding $\mathcal{O}$ constraint (resp. mbb) of the query conjoined with the discriminator for each subtree. This reduces the number of subtrees searched (see Example 2). In our experiments it improves index accesses by $\approx 2\%$, and reduces the number of results by $\approx 3\%$ for lines and $\approx 12\%$ for polygons for both R-trees and O-trees.

Table 1(a) shows the comparison for line data, giving the the number of index accesses, results and leaf nodes which include an answer, the sum of index and results accesses as well as the sum of index and leaf node accesses. For line

Lines	Lines	R-Tree					O-Tree				
T_1	T_2	Index Acc.	Results	Leaves (hits)	Index + Results	Index + Leaves	Index Acc.	Results	Leaves (hits)	Index + Results	Index + Leaves
100	100	76	100	66	176	142	84	100	54	184	138
500	500	898	517	474	1415	1372	892	509	502	1401	1394
1000	1000	2246	1085	1006	3331	3252	2354	1046	1124	3400	3478
5000	5000	17594	7026	6228	24620	23822	20432	6075	6986	26507	27418
10000	10000	47244	18253	16082	65497	63326	53778	14427	16620	68205	70398

Poly gons	Poly gons	R-Tree					O-Tree				
T_1	T_2	Index Acc.	Results	Leaves (hits)	Index + Results	Index + Leaves	Index Acc.	Results	Leaves (hits)	Index + Results	Index + Leaves
100	100	86	5215	84	5301	170	290	4019	288	4309	578
500	500	2466	113629	2444	116095	4910	7570	85652	7316	93222	14886
1000	1000	10124	467219	10048	477343	20172	28384	349576	27556	377960	55940
5000	5000	260352	11465491	258574	11725843	518926	710478	8607313	687470	9317791	1397948

Poly gons	Lines	R-Tree					O-Tree				
T_1	T_2	Index Acc.	Results	Leaves (hits)	Index + Results	Index + Leaves	Index Acc.	Results	Leaves (hits)	Index + Results	Index + Leaves
100	100	130	1474	118	1604	248	186	696	144	882	330
500	500	2466	34245	2236	36711	4702	4540	15137	3378	19677	7918
1000	1000	8322	136522	7732	144844	16054	16076	57563	11598	73639	27674
5000	5000	177848	3396029	163196	3573877	341044	340786	1444966	242184	1785752	582970

Table 2. R-tree versus O-tree join for (a) line$\bowtie$line, (b) polygon$\bowtie$polygon and (c) polygon$\bowtie$line

data in general the O-tree improves upon the R-tree. The greater selectivity made possible by the extra discriminator information is able to reduce the number of index block accesses required. The advantage is increased by the reduced number of results found in the O-tree. In the one case the O-tree does not improve the R-tree the O-tree is 25% higher. The extra index accesses from the height outweigh the small gains in terms of less answers.

Table 1(b) shows the comparison for polygon data. Here we see that the O-trees require more index accesses even when the height of the O-tree is equal to the R-tree. This results from the fact that the tree is wider and, for this polygon data, there are often large overlaps in the discriminators higher in the tree. But the extra discriminating information reduces the results substantially. In the case where retrieving each polygon in the result requires another block access the O-tree uniformly beats the R-tree. If polygons are clustered and/or small, the R-tree is better.

5.2. Join comparison

We now compare the performance of R-trees and O-trees in equi-joins (constraint joins with $c \equiv true$) of two different relations of the same size. The equi-join looks for pairs of objects whose intersection is non-empty. We only discover the number of candidate pairs, that is those objects whose bounding boxes or $\mathcal{O}$ discriminators intersect. Another phase would be required to check if the objects themselves (lines or polygons) actually intersect. For lines this is straightforward, but for polygons it is non-trivial. In these experiments we do not use the propagation solvers since they cannot generate new information in the R-tree case and rarely do in the O-tree case.

Table 2(a) shows the comparison for joining line data with line data. The table shows the number of entries in each of the two join relations, the number of index block accesses required by each method plus the results as well as number of leaf nodes which have hits. As illustrated here, for a join query the number of results can be less than the number of leaves with hits since a leaf hit in each relation may only produce one result. Note that the number of data accesses required in the worst case is still greater than the number of results.

Surprisingly O-trees are outperformed by R-trees even though the intersection query data above indicates better performance on index accesses for lines. This is because in join queries, search is directed by discriminator constraints in contrast to intersection queries where the query constraint closely approximates one particular object, and is thus usually substantially smaller.

Table 2(b) shows the comparison for joining polygon data with polygon data. As seen previously the extra discriminator information available in O-trees is not so useful in separating polygons. Hence the O-tree usually requires substantially more index block accesses than the corresponding R-tree. But in these experiments the extra discriminating power of the O-tree substantially reduces the number of results found. In the worse case for data access the advantage in results for the O-tree means that overall it performs better than the R-tree.

Table 2(c) shows the comparison for joining polygon data with line data. The tradeoff is the same as for the polygon-polygon join. The O-tree requires more index accesses, but improves the number of results substantially.

6. Related Work

The closest related work to height-balanced constraint search trees is another general framework for search trees called Generalized Search Trees (GiSTs) defined by Hellerstein *et al.* [3]. This index structure provides a structure for an extensible set of queries and data types. GiSTs are height-balanced search trees where discriminators are arbitrary predicates and data items are tuples that give values to all variables occurring in any predicate. They are defined by 6 methods, Consistent, Union, Compress, Decompress, Penalty and PickSplit. Consistent, Union and PickSplit correspond to satisfiable, union and split of the HCST framework, while Penalty is related to measure. Compression is not considered in HCSTs.

Though developed independently the frameworks are very similar, the advantages of HCSTs over GiSTs arise from the understanding of predicates and data items as constraints and the, then obvious, use of sophisticated constraint solving to define satisfiable. Since the HCST framework only requires partial algorithms it collects constraints during search down the tree to support incomplete constraint solving. This does not occur in GiSTs. The restriction to tuples is removed in HCSTs where data items are arbitrary constraints, and do not need to involve, let alone fix, all variables.

In some sense constraint search trees answer one of the questions asked in the conclusion of [3], by defining "indexability", what structures are amenable to indexing, as those with feasible constraint solvers. The degree of incompleteness of the constraint solver determines how well indexed a structure is.

O-trees are certainly also definable in the GiST framework, but to the best of our knowledge they have not been previously empirically studied. They can be seen as a form of P-tree [5]. P-trees are defined by mapping an object to the lower and upper bounds on any fixed number k of dimensions (in our case x, y, v and w) and then storing the result in a k dimensional R-tree. This approach immediately loses the connection between the dimensions that is vital for strong constraint solving behaviour. [5] also only gives a theoretical discussion of P-trees, and gives no empirical comparison.

7. Conclusion and Future Work

Height-balanced constraint search trees are the natural form of constraint search tree for storing large amounts of constraint data. The constraint viewpoint makes the algorithms simple to express and implement, and illustrates a natural logical view of search. It also immediately defines efficient index structures in terms of available constraint solvers.

O-trees are a simple form of HCSTs defining a spatial index structure. Our motivation for examining O-trees arose from earlier work on constraint solving for unit two-variable per inequality constraints [4]. Comparing O-trees versus R-trees on 2-d data it seems that the extra discriminating ability of O-trees is usually overridden by the reduction of fan-out in the resulting trees because of the larger size of $\mathcal{O}$ discriminators. For line intersection queries O-trees are superior, and for join queries the reduction in join candidates determined by the O-tree may be advantageous.

We intend to investigate more efficient methods of storing $\mathcal{O}$ discriminators. Constraints from domain $\mathcal{O}$ require at most 8 numbers to represent, but many require less. Of those illustrated in Figure 1 only D requires 8 numbers to store, each of A, B, C, E, F and G require only 4 numbers. Hence we could store $\mathcal{O}$ constraints as an 8 bit guide to which bounds are present plus a 32 bit number for each such bound. In this manner any line constraint will be stored in $8+4\times 32$ bits rather than 8×32 bits. In fact, right-angled triangle shapes can be stored in $8 + 3 \times 32$, less than the size of the bounding box. At least for external nodes, this storage technique should significantly increase the number of constraints that can fit in a node. Another possibility is to restrict $\mathcal{O}$ constraints in the tree to have no more than 4 sides, thus removing the fan-out problem.

We have concentrated on 2-d data. We intend to investigate O-trees for higher dimensions, where for example we also need to represent UTVPI constraints in a compact manner. Another extension is to use more powerful bounds propagation solvers, that handle non-linear constraints, in order to efficiently support much more complex forms of intersection queries.

References

[1] J. Bentley. Multidimensional binary search trees used for associative searching. *CACM*, 18(9):509–517, 1975.

[2] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, 1984.

[3] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proc. of the 21th VLDB Conference*, Zurich, Switzerland, 1995.

[4] J. Jaffar, M. J. Maher, P. Stuckey, and R. Yap. Beyond finite domains. In *Proceedings of the International Workshop on Principle and Practices of Constraint Programming*, number 874 in LNCS, pages 86–93, Orcas Island, Washington, May 1994. Springer-Verlag.

[5] H. V. Jagadish. Spatial search with polyhedra. In *Proceedings of the IEEE Int. Conf on Data Engineering*, pages 311–319, 1990.

[6] K. Marriott and P. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.

[7] R. Nelson and H. Samet. A Consistent Hierarchical Representation for VectorData. *Computer Graphics*, 20(4), August 1986.

[8] P. Stuckey. Constraint Search Trees. In *Logic Programming: Proc. of the 14th International Conference*, pages 301–315, Cambridge, MA, July 1997. MIT Press.