Constraint Search Trees

Peter J. Stuckey
Department of Computer Science
University of Melbourne
Parkville 3052, Australia
pjs@cs.mu.oz.au

Abstract

This paper defines constraint search trees, a general tree data structure for storing and accessing items with constraints as keys. Constraint search trees can mimic binary search trees, radix search trees, k-d trees, R-trees and many other kinds of search trees. The unifying framework of constraint search trees immediately suggests new kinds of spatial data indexing techniques, as well as showing how to generalize intersection queries in spatial data structures for arbitrary shapes defined by constraints. Constraint search trees provide a data structure for storing sets (or disjunctions) of constraints. Hence they provide a useful data structure for implementing constraint databases. We define efficient algorithms for constraint database operations such as constraint selection, join and subsumption applied to constraint search trees.

1 Introduction

Search trees are a fundamental data structure of computer science, providing a way of storing collection 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 [2], B-trees and R-trees [7]. Constraint search trees defined in this paper abstract the fundamental behaviour of a search tree in terms of constraints. Hence they can be used to mimic all of the above kinds of trees.

Search trees are heavily used in many computer science applications. Database application make heavy use of indexing methods based on search trees, in particular geographic information systems use many forms of search trees for spatial data. Search trees also provide efficient algorithms for computer graphics (see e.g. [5]).

General algorithms for constraint search trees provide insight into the operations that underly all search trees as well as how methods for particular search trees can be extended to other kinds of search tree. The contributions of this paper are

- We define *constraint search trees* a general tree data structure for storing and accessing constraints. Complex constraint domains usually only provide partial tests for satisfiability and implication of constraints, so constraint search trees are designed to work with such partial algorithms.

- We show how the notion of constraint search tree immediately suggests new indexing approaches to spatial data using constraint domains (Example 10).
- Constraint search trees provide an excellent data structure for constraint database implementations. We give algorithms for constraint database operations such as select, join and subsume.
- We show how the constraint database operations defined for constraint search trees can suggest new approaches to relational database evaluation (Example 13).

The closest related work to this paper is the recent paper of Sexton [11]. He independently makes the important realization that indexes can be any logical statements that describe the objects stored in their corresponding tree. He develops a formalism for indexes based on an approximation lattice and sketches how this can be used to evaluate constraint select, constraint join and sort queries on indexed data. Constraint techniques are used solely to prune search, where they are restricted to where their operation is complete. Both Bressan *et al.* [3] and Goldstein *et al.* [6] show how, using constraint propagation techniques, constraint select queries can be more efficiently evaluated. Goldstein *et al.* give detailed empirical results demonstrating the benefits that can be achieved. None of the above work investigates storing complex constraints in trees.

The remainder of the paper is organized as follows. In Section 2 we define constraint search trees and their fundamental search operations and give examples relating them to other kinds of search trees. In Section 3 we discuss building constraint search trees, while in Section 4 we explain important factors influencing the efficiency of constraint search trees. In Section 5 we show how constraint search trees can be used for constraint database evaluation.

2 Constraint Search Trees

Constraint search trees store objects using a constraint key. For simplicity in this paper we ignore the rest of the object and concentrate on storing and accessing only the constraint keys. Constraint search trees store constraints in a manner analogous to other search trees. The ordering component of a constraint search tree is by implication. For two constraints c and d either $c \rightarrow d$, or the implication does not hold. This ordering behaviour is used to construct binary trees which can be searched efficiently.

Constraint search trees make use of two languages of constraints. The language of constraint items c stored in the tree, and the language of discriminator constraints d used to differentiate constraints.

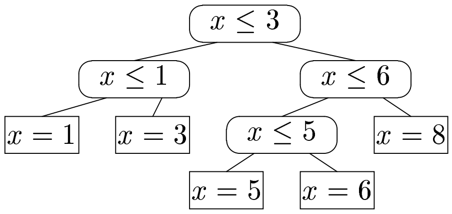

Figure 1: A constraint search tree for integers

A *constraint search tree* is made up of two kinds of nodes: external nodes $ext(C)$ containing simply a (possibly empty) set of constraints C, and internal nodes $int(d, t, u)$. In an internal node

- d is a discriminator constraint used to separate the children of the node;
- t is a tree containing constraints that imply the discriminator; and
- u is a tree containing constraints that do not imply the discriminator.

In section 5 we will consider a weaker version of constraint search trees where we relax the restriction on the u subtree.

We begin by relating constraint search trees to simple binary search trees over integers.

Example 1. Consider a search tree over integers. An integer i can be thought of as a constraint over some single fixed variable x of the form $x = i$. This is the language of the constraint items. The language of discriminators is simple inequalities $x \leq i$ where i is some integer. A constraint search tree storing the integers $1, 3, 5, 6, 8$ is shown in Figure 1. A round cornered box is an internal node, while a rectangular box is an external node. Each item in the tree t implies the discriminator constraint d, and in this example each item in the tree u implies the negation of the discriminator. For example $x = 5 \rightarrow \neg(x \leq 3)$, $x = 5 \rightarrow x \leq 6$ and $x = 5 \rightarrow x \leq 5$.

Imagine searching for the item $x = 4$. The constraint does not imply $x \leq 3$ so we take the u branch, it implies $x \leq 6$ so we take the t branch, and again implies $x \leq 5$. Searching the external node we fail to find $x = 4$ and hence the search fails. $\diamond$

Constraint items are not stored in internal nodes[1] because this muddles the difference between the items and discriminators. We can however achieve the same effect as shown in the following example.

[1]Hence constraint search *tries* might be a more accurate name.

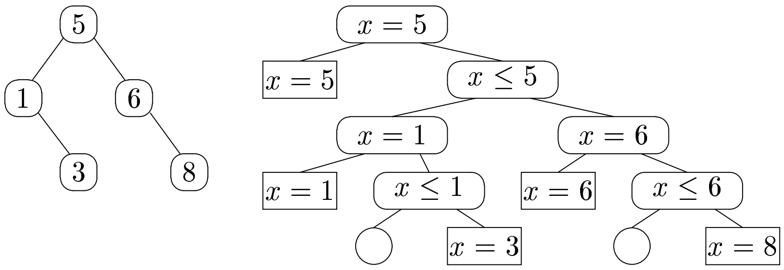

Figure 2: A binary search tree and corresponding constraint search tree

Example 2. We can obtain a constraint search tree representing a binary search tree by adding discriminators like $x = 4$ to the language of discriminators. A binary search tree for $\{1, 3, 5, 6, 8\}$ and its corresponding constraint search tree are shown in Figure 2, where a circle denotes an empty external node $ext(\{\})$. One may argue that the constraint search tree more explicitly shows the tests that occur in the original binary search tree. When searching for an item we first test for equality and then branch depending upon whether it is larger or smaller. $\diamond$

The above constraint search trees show examples where only one constraint resides in each external node. This is not required in general. For example consider the following case:

Example 3. A rectangle with southwest corner (d_1, d_2) and northeast corner (d_3, d_4) can be represented by a *rectangle constraint* on variables x, y as:

$$d_1 \le x \wedge x \le d_3 \wedge d_2 \le y \wedge y \le d_4$$

We can build constraint search trees for storing rectangle items, using rectangle discriminators, which corresponds to the R-tree family [7, 10]. A rectangle constraint implies another if the first rectangle sits entirely inside the second.

Consider the rectangles shown at the left of Figure 3. The constraints represented by rectangles A to G are stored in a constraint search tree using discriminator rectangles H and I as shown in the right of Figure 3. Notice that rectangle E neither implies I or its negation, since they overlap.

The external nodes here hold more than one constraint item because we choose not to discriminate them further. $\diamond$

To formally define constraint trees we need a formal model of the constraints over which they operate. A *constraint domain* $\mathcal{A}$ consists of: a signature defining the language of constraints (the function and relation symbols), and an interpretation $M_{\mathcal{A}}$ defining the meaning of each function

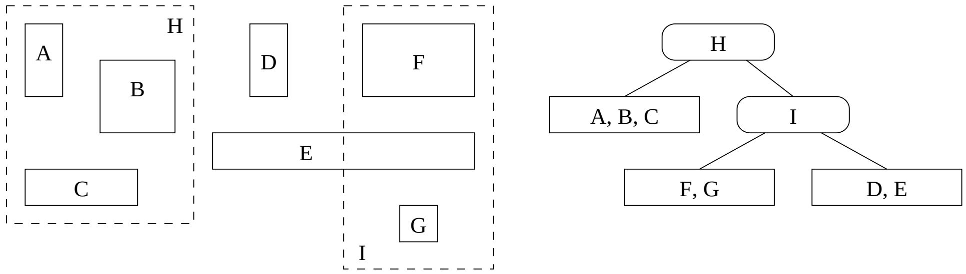

Figure 3: A constraint search tree for rectangles

and relation symbol in the signature, together with a set of first order formulae $F_{\mathcal{A}}$ defining the acceptable *constraints* of the domain. We assume the $F_{\mathcal{A}}$ is closed under conjunction.

In practice for making use of constraint search trees we need 2 functions for handling constraints. A function $\mathsf{satisfiable} :: F_{\mathcal{A}} \mapsto \{true, false, unknown\}$ which determines whether a constraint is satisfiable or not, and a function $\mathsf{implies} :: F_{\mathcal{A}} \times F_{\mathcal{A}} \mapsto \{true, false, unknown\}$ which determines whether one constraint implies another.

The functions are satisfiable and implies are meant to reflect the meaning of constraint given by the interpretation $M_{\mathcal{A}}$. They can be *incomplete*, that is answer *unknown*, but must satisfy the following correctness conditions:

$$\begin{aligned}
\text{if } \mathsf{satisfiable}(c) &= true \text{ then } M_{\mathcal{A}} \models \tilde{\exists} c \\
&= false \text{ then } M_{\mathcal{A}} \models \neg\tilde{\exists} c \\
\text{if } \mathsf{implies}(c_1, c_2) &= true \text{ then } M_{\mathcal{A}} \models \tilde{\forall} c_1 \rightarrow c_2 \\
&= false \text{ then } M_{\mathcal{A}} \models \neg(\tilde{\forall} c_1 \rightarrow c_2)
\end{aligned}$$

We say satisfiable and implies are *complete* if they always answer either *true* or *false*.

The language of constraint items and the discriminator language are both subsets of $F_{\mathcal{A}}$. The algorithms in this paper assume that the language of discriminators is closed under negation. Note that discriminators for the constraint search tree for rectangles used in Example 3 are not closed under negation. We can remove this restriction by complicating the algorithms somewhat.

Example 4. An example constraint domain is $\mathcal{Z}$, the integers, with the functions (constants) $\{0, 1, +, *, (unary)-\}$ and the constraint relations $\{=, \leq, <\}$, and the usual interpretation. Let the constraint language $F_{\mathcal{Z}}$ be arbitrary un-quantified conjunctions of integer constraints.

We shall allow constraint items to be any constraint in $F_{\mathcal{Z}}$ but restrict the discriminator language to constraints of the form $v \leq i$ or $i \leq v$, where v is a variable and i is an integer.

Any function satisfiable for $\mathcal{Z}$ is necessarily incomplete because the problem of integer satisfiability is undecidable. One efficient approach to determining partial satisfiability of integer constraint c is using bounds consistency (see e.g. [12]) to determine for each variable v in c a range $[l..u]$ which it must lie within in all solutions of c. If one of these ranges is empty satisfiable(c) returns $false$, if each is a singleton satisfiable(c) returns $true$, otherwise it returns $unknown$.

A general implication algorithm is again undecidable. But we can define an efficient implication algorithm for some special cases using the same bounds consistency approach as follows: implies($c, v \leq i$) is $true$ if the range calculated for v in c is $[l..u]$ and $u \leq i$, and implies($c, i \leq v$) is $true$ if the range calculated for v in c is $[l..u]$ and $i \leq l$. In other cases the implication tester returns $unknown$. $\diamond$

Constraint search trees support two kinds of search: find(T, c) which finds if a constraint c from the item language appears in the tree T, and conjoin(T, c) which returns constraints which are mutually satisfiable with an arbitrary constraint $c \in F_{\mathcal{A}}$.

The algorithm for find simply traverses the tree searching for whether constraint c is in the tree using the discriminator information. Note that this is an exact match, it does not find constraints equivalent to c because of the partialness of implies. If implies is complete we can straightforwardly modify find to find equivalent constraints otherwise an equivalence search has to be closer in form to conjoin.

```
find(T, c)
    case T of
    ext(C):        if (c ∈ C) return true
                   return false
    int(d, t, u):  if implies(c, d) return find(t, c)
                   return find(u, c)
```

Example 5. Consider the (partially illustrated) tree for constraint domain $\mathcal{Z}$ shown in Figure 4. Searching for the item $c \equiv x \leq 0 \wedge 4y^2 + z^2 = 4$ can be performed as follows. We first determine using bounds propagation the ranges of the variables $x :: (-\infty..0]$, $y :: [-1..1]$ and $z :: [-2..2]$. Now implies($c, z \leq 2$) clearly holds so we follow the t branch. implies($c, x \geq 4$) does not hold so we follow the u branch. The implication holds for $y \leq 1$ so we follow the t branch. At the external node the constraint c is found. $\diamond$

Conjoin or overlap queries search all the tree to find constraints that are possibly satisfiable when conjoined with the query constraint. In this search we always need to traverse the u branch, but can avoid searching the t branch if the constraint c implies the negation of the discriminator.

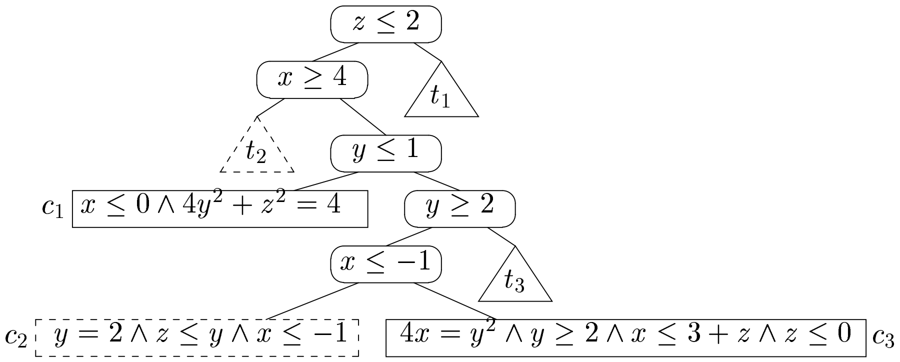

Figure 4: A constraint search tree for arbitrary integer constraints

Importantly as we traverse the tree we collect the discriminator constraints which are known to hold 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.

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

Example 6. Imagine searching for constraints that conjoin with $c \equiv x^2 + y^2 \leq z^2 \wedge -5 \leq x, y, z \leq 5$. The bounds calculated for each variable are $x :: [-5..5]$, $y :: [-5..5]$, and $z :: [-5..5]$.

The query $\mathsf{conjoin}(T, c)$, where T is the tree (partially) illustrated in Figure 4, proceeds as follows. The current bounds do not imply the negation of discriminator $z \leq 2$ so both subtrees of the root node must be visited. Visiting the t subtree the constraint is modified to include $z \leq 2$. The bounds calculated for $c \wedge z \leq 2$ are $x :: [-2..2]$, $y :: [-2..2]$, and $z :: [-5..2]$. At the second internal node the negation of the discriminator $x \geq 4$ is clearly implied by the bounds information hence we do not have to visit tree t_2.

Examining the internal node with discriminator $y \leq 1$ both subtrees need to be explored. Exploring the t subtree with the constraint $c \wedge z \leq 2 \wedge y \leq 1$ the answer $c_1 \equiv x \leq 0 \wedge 4y^2 + z^2 = 4$ is found to be possibly satisfiable with c, that is $\mathsf{satisfiable}(c \wedge z \leq 2 \wedge y \leq 1 \wedge c_1)$ returns *unknown* hence c_1 is collected as an answer.

After exploring the t subtree we return and explore the u subtree which has the discriminator $y \geq 2$. Again both subtrees need to be explored. We visit the t subtree with constraint $c \wedge z \leq 2 \wedge y \geq 2$. The bounds calculated for this constraint are $x :: [0..0]$, $y :: [2..2]$, and $z :: [-2..2]$. Only one value is available for x and y, hence the query is now almost a point query (at least in this part of the subtree). The negation of the discriminator $x \leq -1$ is implied hence we do not visit the tree containing $c_2 \equiv y = 2 \wedge z \leq y \wedge x \leq -1$.

Visiting the u subtree containing $c_3 \equiv 4x = y^2 \wedge y \geq 2 \wedge x \leq 3 + z \wedge z \leq 0$, we find $\mathsf{satisfiable}(c \wedge z \leq 2 \wedge y \geq 2 \wedge c_3)$ returns $false$. Hence c_3 is not collected as an answer. Search continues examining t_3 and t_1.

Notice that, for the original constraint c none of the negations of the discriminators is implied even with a complete implication function. ◇

The conjoin algorithm works well with incremental backtracking constraint solving methods usual in constraint programming. This is because, for $\neg(\tilde{\forall} c \to \neg d) \;\leftrightarrow\; \exists c \wedge d$. Hence we can replace the test $\mathsf{implies}(c, \neg d) \neq true$ by $\mathsf{satisfiable}(c \wedge d) \neq false$.[2]

If the constraint solver holds c initially, to traverse the tree we simply add the discriminator d to the solver and test for satisfiability. If this does not result in $false$ we visit the t subtree recursively, and then visit the u subtree. Otherwise we remove d and visit the u subtree. When checking an external node each constraint is added to the solver in turn, checking for satisfiability. Backtracking is used to restore the solvers state.

The conjoin algorithm is simple but very powerful. For example implicitly it defines how to do complicated constraint searches in spatial data structures such as k-d trees [2].

Example 7. A 3-d tree storing points (d_1, d_2, d_3) is a constraint search tree storing constraints of the form $x = d_1 \wedge y = d_2 \wedge z = d_3$ where discriminators are of the form $v \leq i$ and $i \leq v$. Let us build a 3-d tree for integers similar to that shown in Figure 4 by replacing the constraints c_1, c_2 and c_3 by respectively $(0, 1, 2)$, $(-2, 2, 1)$ and $(0, 2, 2)$. Searching the 3-d tree for points within the shape defined by constraint $c \equiv x^2 + y^2 \leq z^2 \wedge -5 \leq x, y, z \leq 5$ occurs exactly as in Example 6 above except the satisfiability tests are easier. The answer $(0, 1, 2)$ is found. Importantly the point $(-2, 2, 1)$ and the whole subtree t_2 are never considered in the search, even though the query constraint c does not imply the negation of any discriminator. ◇

Both Bressan *et al.* [3] and Goldstein *et al.* [6] show how using constraint methods can be used to improve conjoin queries in existing data structures such as B-trees and R-trees. Bressan *et al.* sketch how to use a constraint solver for $\mathcal{Z}$ to improve access to a B-tree, while Goldstein *et al.* develop a clipping algorithm (a kind of floating point bounds consistency technique) which can be used to improve conjoin queries defined by conjunctions of linear real inequalities applied to R-trees, and give compelling empirical

[2] We could have presented the algorithm using satisfiable rather than implies but the implies function is required in any case, and this presentation more faithfully matches our view that implication is the tool for controlling the tree search.

evidence for its use. Sexton [11] also illustrates how propagation methods can improve conjoin queries.

3 Building constraint search trees

Up to this point we have said nothing about how constraint search trees can be created. We now give an algorithm for insertion into a constraint search tree. Other approaches to building constraint search trees, such as bottom-up approaches are equally valid. Because constraint items are stored at the leaves, deletion from constraint search trees is straightforward.

```
insert(T, c)
    case T of
    ext(C):       return split(ext(C ∪ {c}))
    int(d, t, u): if implies(c, d) return int(d, insert(t, c), u)
                  else return int(d, t, insert(u, c))
```

The splitting function split is the key to insertion. It determines how to split a set of constraints C, represented as an external node $ext(C)$. First note that we do not have to split a node, we may in choose to wait until it grows larger than some size. Second note that because the language of discriminators is restricted it may be impossible to split two different constraints. For example each of the constraints corresponding to the shapes in Figure 5 is not discriminable using just constraints of the form $v \leq i$ and $i \leq v$ if they are centred at the same point, as shown on the right.

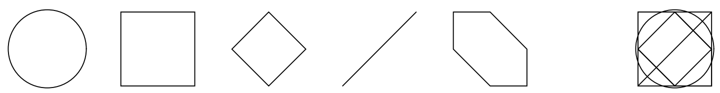

Figure 5: A collection of non-discriminable shapes

The key to splitting two constraints is finding a discriminator for them if possible. A discriminator d can discriminate two constraints c_1 and c_2 in two different ways. Assume $\mathsf{implies}(c_1, d)$, then either (a) $\mathsf{implies}(c_2, \neg d)$ or (b) $\mathsf{implies}(c_2, d) \neq true$ and $\mathsf{implies}(c_2, \neg d) \neq true$. Discrimination of type (a) implicitly gives more information about c_2.

Example 8. We can define split for $\mathcal{Z}$ for sets of the form $\{c_1, c_2\}$ using the procedure below. We can repeatedly apply it to split larger sets (although cleverer algorithms are possible).

Determine the bounds for each variable v in relation to c_1, that is $v :: [l_v^1..u_v^1]$ and c_2, $v :: [l_v^2..u_v^2]$. If the bounds $[l_v^1..u_v^1]$ and $[l_v^2..u_v^2]$ are identical for all variables, then c_1 and c_2 are not discriminable, return $ext(\{c_1, c_2\})$.

Otherwise if there exists variable v where $[l_v^1..u_v^1]$ and $[l_v^2..u_v^2]$ do not intersect then we can build a discriminator of type (a). Assuming $u_v^1 < l_v^2$ then return $int(v \leq u_v^1, ext(\{c_1\}), ext(\{c_2\}))$. Other cases are similar.

Otherwise choose a variable v where $[l_v^1..u_v^1]$ is not identical to $[l_v^2..u_v^2]$. In the case that $l_v^2 < l_v^1$ we can return $int(l_v^1 \leq v, ext(\{c_1\}), ext(\{c_2\}))$. Other cases are similar. ◇

Example 9. Imagine inserting constraint $c_4 \equiv x + 3y \leq 3 \wedge z \leq 2 \wedge 0 \leq x \wedge 0 \leq y$ into the constraint search tree shown in Figure 4. The bounds for this constraint are $x :: [0..3]$, $y :: [0..1]$, $z :: (-\infty..2]$ so the search in insert takes us to the node containing $c_1 \equiv x \leq 0 \wedge 4y^2 + z^2 = 4$. The bounds for this constraint are $x :: (-\infty..0]$, $y :: [-1..1]$, $z :: [-2..2]$. Each of the ranges overlap with those of c_1, hence we can only achieve type (b) discrimination for example using the discriminator $0 \leq y$ we obtain the tree $int(0 \leq y, ext(\{c_4\}), ext(\{c_1\}))$. ◇

4 Efficient constraint search trees

For efficient usage a constraint tree needs to be able to perform the test implies$(c, \neg d)$ efficiently. Obviously for arbitrary constraints these tests are not necessarily very efficient. By appropriate choices of the constraint language and discriminator language we can however obtain efficient instances.

Other examples of search trees, such as binary search trees, radix search trees, k-d trees and R-trees all correspond to choices of item language and discriminator language where the implication test is very efficient.

Given any constraint domain which has an efficient test for implication we can build an indexing methodology from it using constraint search trees.

Example 10. Consider the constraint domain of real numbers with a language consisting of conjunctions of unit two-variable-per-inequality (UTVPI) inequalities, of the forms $ax + by \leq d$ and $ax + by < d$ where $\{a, b\} \subseteq \{-1, 0, 1\}$. Let the constraint items be any such constraint, while discriminators are single UTVPI inequalities.

Such constraints have an efficient incremental satisfiability test and implication test for discriminators (both $(O(n^2)$ where n is the number of inequalities in the constraint item) based on transitive closure (see [8][3]).

Analogous to the use of bounding boxes in spatial data structures we can use UTVPI constraints to define a bounding structure for an arbitrary k dimensional object. Note that UTVPI constraints can discriminate each of the constraints represented in Figure 5. We can then use it as a key for storing these objects in a constraint tree. Because the constraint solver for UTVPI constraints is incremental and holds all the UTVPI implications of a constraint it can be used very efficiently for constraint tree search. ◇

[3] Although the algorithms there are for the integer case.

Given the implication tests are efficient enough the next important feature of a constraint search tree for efficiency is its shape. The shape of a constraint search tree built by repeated insertions depends heavily on the split function. If the constraint domain and distribution of constraint items implies that inserting a random constraint into a constraint search will result in the new constraint being added to any leaf node with more or less equal probability then we obtain the usual logarithmic height average case behaviour. But for many constraint domains and distributions of constraints this is not the case. For example when implies always returns *unknown* then no discrimination is possible. Even if implies is complete we cannot expect logarithmic behaviour for arbitrary choices of item languages and discriminator languages

Example 11. Consider constraint search trees over real constraints where constraint items are each planes in 3 dimensions of the form $ax+by+cz=d$, and discriminators are of the form $a'x+b'y+c'z \leq d'$. For a random item c and discriminator d the probability that $\mathsf{implies}(c, \neg d) = unknown$ is 1 since every plane intersects almost every other plane. Hence although we can discriminate any two non-equivalent items we expect any constraint search trees built from n random inserts to have height n. $\diamond$

Although in the worst case the height of a constraint search tree is $O(n)$ where n is the number of constraint items it contains. In practice constraint search trees are likely to be logarithmic in height if any discriminator has a reasonable chance of discriminating any two constraints.

Obviously for many choices of constraint domain we can do better than $O(n)$, in particular those corresponding to binary search trees, k-d trees B-trees and R-trees. For example given a constraint domain where variables range over $0..2^k-1$ we can construct a strong insertion routine analogous to those use for radix search trees (or quad- and oct-trees) that generates a search tree which has height $O(n_v k)$, where n_v is the number of variables appearing in constraints in the tree.

5 Constraint search trees for constraint databases

A constraint search tree represents a set (or disjunction) of constraints. This is exactly what is required for representing the value of a constraint relation in a constraint database [9]. In this section we give algorithms for constraint database operations applied to constraint search trees. Where most work in constraint databases assumes complete algorithms for handling constraints, no such assumption is made here. We do make the assumption that the item language is closed under conjunction with any constraint $c \in F_{\mathcal{A}}$ so that the trees produced only contain legitimate items.

The algorithms presented do not guarantee that every constraint in the u subtree of tree $int(d, t, u)$ does not imply the discriminator d. Hence in general they produce *weak* constraint search trees. We can define more complicated algorithms that avoid this behaviour. But in any case all the algorithms presented, except find, are correct for *weak* constraint search trees.

The problem of *constraint selection* is, given a set of constraints C and a selection condition c, build the set $\{c' \wedge c \mid c' \in C, \mathsf{satisfiable}(c' \wedge c)\}$. We define the function $\mathsf{select}(T, c, true)$, performing a selection c on a set of constraints represented by constraint search tree T and returning a constraint search tree containing the answer, below. The third argument p, initially *true* stores the path of discriminators followed.

```
select(T, c, p)
    if implies(p, c) return T
    case T of
    ext(C):       return split(ext({c' ∧ c | c' ∈ C, satisfiable(c' ∧ c)}))
    int(d, t, u): t' := ext({})
                  if (implies(c, ¬d) ≠ true) t' := select(t, c ∧ d, p ∧ d)
                  u' := select(u, c, p)
                  if (t' = ext({})) return u'
                  elseif (u' = ext({})) return t'
                  else return int(d, t', u')
```

select uses the information of the selection condition to narrow the search so that not all constraints in the tree need to be examined. When the path constraint p implies c then every constraint within the tree T implies c hence the whole tree can be returned. Otherwise a recursive traversal of the tree ensues. When a internal node no longer discriminates, that is one of child trees is empty, the node is removed.[4]

Example 12. Consider a constraint selection upon the tree partially illustrated in Figure 4 with the constraint $c \equiv x \geq 4$. Initially p is *true*. Now the negation of $z \leq 2$ is not implied hence we have to visit both subtrees. Visiting the t subtree, p is now $z \leq 2$. Again the negation of $x \geq 4$ is not implied by c, so we visit the t subtree with p equal to $z \leq 2 \wedge x \geq 4$. We find that $\mathsf{implies}(p, c)$, thus the whole subtree is collected. Execution continues visiting the entire subtree rooted with discriminator $y \leq 1$. None of the constraints shown is satisfiable with $x \geq 4$ hence the only answers come from constraints in t_3 and t_1. Suppose t_1 contained the constraint $x + z \leq 3$, then the result of the selection $x + z \leq 3 \wedge x \geq 4$ implies that $z \leq 2$ and hence should be in the t subtree of the root. It will appear in the u subtree, hence the resulting tree is weak. $\diamond$

The problem of *constraint join* is, given two set of constraints C_1 and C_2 and a join condition c, build the set $\{c_1 \wedge c \wedge c_2 \mid c_1 \in C_1, c_2 \in C_2, \mathsf{satisfiable}(c_1 \wedge c \wedge c_2)\}$. We define the function $\mathsf{join}(T_1, c, T_2)$ for a constraint join on two trees T_1 and T_2 with condition c which returns a tree as follows:

[4] We could use the same device in other operations below but omit it for simplicity.

```
join(T_1, c, T_2)
    case T_1 of
    ext(C):        return setselect(select(T_2, c, true), C)
    int(d, t, u):  t' := ext({})
                   if (implies(c, ¬d) ≠ true) t' := join(t, c ∧ d, T_2)
                   u' := join(u, c, T_2)
                   return int(d, t', u')
setselect(T, C)
    case T of
    ext(C'):       C'' = {c' ∧ c | c ∈ C, c' ∈ C', satisfiable(c' ∧ c)}
                   return split(ext(C''))
    int(d, t, u):  C_t := {c ∧ d | c ∈ C, implies(c, ¬d) ≠ true}
                   t' := setselect(t, C_t)
                   u' := setselect(u, C)
                   return int(d, t', u')
```

join searches each branch of the first tree T_1 adding the discriminator constraints found in the path to an external node. Then it uses this constraint to simplify T_2 to the relevant parts for this external node of T_1. Then it performs a set-wise constraint selection setselect of this simplified tree with respect to the constraints in the external node. Notice that it does perform true set-at-a-time computation since each simplification of T_2 is made with respect to all constraints in one external node.

Algorithms for other constraint database operations such as computing the union of two constraint relations and computing the difference of two constraint relations can be built analogously.

Example 13. One of the advantages of the constraint tree viewpoint is that it shows how to efficiently perform some special case computations for very familiar contexts.

Consider the following SQL query on two relations of pairs of integers $P(x, z)$ and $Q(y, t)$ stored in B-trees using a lexicographic ordering.

```
SELECT  P.x, P.z, Q.y, Q.t
FROM    P, Q
WHERE   P.x = Q.y * P.z
```

The query corresponds to the constraint join $P(x, z) \bowtie_{x=y*z} Q(y, t)$. All relational database query compilers that I am aware of will perform a cross product of P and Q and then apply the constraint as a selection.

Let us examine how the join algorithm for $\mathcal{Z}^5$ behaves for this query. Constraint items for representing the B-tree are e.g. $x = d_1 \wedge z = d_2$, while discriminators $(l_x, l_z) \leq_l (x, z) \leq_l (u_x, u_z)$ for the lexicographic ordering $\leq_l$

[5]The domain has to be slightly modified to handle the more complicated discriminators correctly.

are represented as $l_x \leq x \wedge (l_x < x \vee l_z \leq z) \wedge x \leq u_x \wedge (x < u_x \vee z \leq u_z)$. For each block (an external node) in the B-tree P with lower lexicographic bound (l_x, l_z) and upper bound (u_x, u_z):

- if ($l_x < u_x$ and $0 \in [l_x..u_x]$) or ($l_x = u_x = 0$ and $0 \in [l_z..u_z]$) then retrieve all tuples for Q;
- if $l_x = u_x = 0$ and $0 \notin [l_z..u_z]$ retrieve tuples in Q where y is 0;
- if $l_x = u_x \neq 0$ and $0 \notin [l_z..u_z]$ retrieve tuples in Q where y is in the range $[min(\{l_x/l_z, l_x/u_z\})..max(\{l_x/l_z, l_x/u_z\})]$;
- otherwise $l_x \leq u_x$ and $0 \notin [l_x..u_x]$ so retrieve tuples for Q where y is in the range $[-max(\{|l_x|, |u_x|\})..max(\{|l_x|, |u_x|\})]$

Retrieved tuples are joined with tuples in the block for P and then have the selection $x = y * z$ applied. Note that the B-tree index for Q can support the range queries efficiently.

For only the first kind of block is the full cross-product with Q computed. Hence we have devised a more efficient method for computing $P(x, z) \bowtie_{x=y*z} Q(y, t)$ in a relational database. $\diamond$

Constraint relations represent disjunctions of constraints hence we are interested in keeping them in a minimal form. As we discover new constraints that form part of the relation it is then of interest to remove constraints that are subsumed by other constraints. A constraint c_1 is *subsumed* by another c_2 if $\mathsf{implies}(c_1, c_2)$ holds in which case $c_2 \leftrightarrow c_1 \vee c_2$. $\mathsf{subsume}$ simplifies a tree T by removing all the constraints within subsumed by c and is called initially with the third argument $true$. The third argument holds the constraints found on the current search path. We can straightforwardly combine an insert operation with subsumption.

```
subsume(T, c, p)
    if implies(p, c) return ext({})
    case T of
    ext(C):        return ext({c' | c' ∈ C, implies(c', c) ≠ true})
    int(d, t, u):  t' := t
                   if (implies(c, ¬d) ≠ true) t' := subsume(t, c, p ∧ d)
                   return int(d, t', subsume(u, c, p))
```

The subsumption algorithm differs from all the others by not conjoining the discriminator constraints to the subsuming constraint, this would be erroneous. The interesting points of the subsumption algorithm are the first line where if c is implied by p then anything stored in the tree T is subsumed by c, hence the whole tree is replaced by an empty external node $ext(\{\})$, and the case when $\mathsf{implies}(c, \neg d)$ in which case nothing in the tree t can be subsumed and hence it does not need to be visited.

Worst case complexities of these operations in terms of the basic operations implies, and satisfiable are $O(n)$ for select and subsume, where n is the number of constraints in the tree, and $O(n \times m)$ for join, where n and m are the number of constraints in T_1 and T_2 respectively. These do not improve on any simple algorithm for such operations. The reason is that in the worst case the output is of this size in any case, for example for join representing a cross product like $P(x) \bowtie Q(y)$ no improvement can be attained. Where the discriminators of the trees and the constraints c are related we can expect far better than worst case behaviour.

Given algorithms for select, join, union, difference, subsume and project[6] we can straightforwardly build a constraint database evaluation that supports using semi-naive evaluation with (pairwise) subsumption based on constraint search trees. The algorithms described provide true set-at-a-time evaluation by making use of the discriminator information.

References

[1] F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(Intervals) revisited. *Procs. Int. Logic Programming Symp.*, Ithaca, 1994. 124–138.

[2] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM* **18**(9), 1975. 509–517.

[3] S. Bressan, A. Sexton, M. Wallace. Implementing Index Data Structures Using Constraint Logic Programming. Presented at *CP96 Workshop on Constraints and Databases*, 10 pages, Cambridge, Massachusetts, 1996.

[4] D. Chan. Constructive negation based on the completed database. *Procs. Fifth Int. Conf. and Symp. on Logic Programming*, Seattle, 1988. 111–125.

[5] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structure. *Computer Graphics* **14**(3), 1980. 124–133.

[6] J. Goldstein, R. Ramakrishnan, U. Shaft, and Jie-Bing Yu. Using Constraints to Query R*-Trees. University of Madison-Wisconsin CS-TR-96-1301, 1996.

[7] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Proc. ACM SIGMOD Int. Conf. on Management of Data* 1084. 47–57.

[8] J. Jaffar, M. Maher, P. Stuckey and R. Yap. Beyond finite domains. In *Procs of PPCP'94*, LNCS 874, Seattle WA, May 1994, 86–94.

[9] P. Kannelakis, G. Kuper, and P. Revesz. Constraint query languages. *JCSS* **51**, 1995. 26–52.

[10] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R$^+$-tree: a dynamic index for multidimensional objects. *Procs 13th VLDB Conf*, Brighton 1987. 507–518.

[11] A. Sexton. Querying Indexed Files. In *Procs of CDB'97*, LNCS 1191, Delphi Greece, January 1997, 263–281.

[12] P. Van Hentenryck. *Constraint satisfaction in logic programming*. MIT Press, 1989.

[6] Which we did not discuss for space reasons.