Data Structures for Generalised Arc Consistency for Extensional Constraints

Ian P. Gent, Ian Miguel, Peter Nightingale
School of Computer Science, University of St Andrews, Scotland, KY16 9SX
{ ipg, ianm, pn}@dcs.st-and.ac.uk
Keywords: Constraint Programming, Constraint Satisfaction, Global Constraints

September 9, 2006

Abstract

We describe the use of two alternative data structures for maintaining generalised arc consistency on table/extensional constraints. The first, the Next-Difference list, is novel and has been developed with this application in mind. The second, the trie, is well known but its use in this context is novel. Empirical analyses demonstrate the efficiency of the resulting approaches.

1 Introduction

Constraint programming is a successful technology for solving a wide variety of combinatorial problems from industry and academia. Examples include scheduling [1], industrial design [5] and combinatorial mathematics [12], to name but a few examples [13]. Constraint programming can be viewed as a two-stage process. First, the problem is characterised, or *modelled* as a set of constraints on decision variables that its solutions must satisfy. Second, a search process is used to find solutions. Typically, this process interleaves the choice of an instantiation of a decision variable with the *propagation* of the constraints to determine the consequences of the choice made.

Constraint technology is typically available in one of two forms. The first is a toolkit embedded in a given host programming language, such as C++, Java or Prolog. Examples include Ilog Solver [7], Eclipse[1], or GECODE[2]. The second is a standalone solver, such as Minion [6]. In either case, a library of constraints is made available with which to model problems. The extensional or 'table' constraint is the basic building block of a constraint library. Since it allows us simply to list the allowed combinations of values to a particular subset of the variables it can be used to express any relation straightforwardly when it might be awkward or cumbersome to do so using the other primitives available in the library.

To illustrate, consider modelling the constraint x 'likes' y on two decision variables x and y, both of which have domains {bill, bert, tom}. It is a simple matter to write down the extension of this constraint (the set of satisfying assignment pairs), e.g. {⟨ Bill, Bert ⟩, ⟨ Bill, Tom ⟩, ⟨ Bert, Tom ⟩}, This represents that Bill likes Bert and Tom, Bert likes Tom, and Tom likes noone. On the other hand attempting to use a collection of other constraints, such as inequalities and implications, to express the same relation is an awkward task.

[1] http://eclipse.crosscoreop.com/eclipse/
[2] http://www.gecode.org

The natural question is why not use extensional constraints alone? The problem with this approach is one of efficiency. An extensional representation of a constraint can become very large and expensive to process, especially as the arity of the constraint and the size of the domains of the variables involved grows. Hence, intensional representations for certain constraints, e.g. the AllDifferent constraint [11] or lexicographic ordering [4], are far more efficient in practice.

Where the extensional representation remains the most effective way to model a facet of a problem it is essential to be able to propagate this type of constraint as efficiently as possible. The naive approach, which is simply to traverse the list of satisfying tuples in search of 'support' for a given assignment, is very expensive. Previously, Lhomme and Régin introduced the *hologram tuple* data structure to improve this process [9]. In this paper, we introduce two new ways of improving the propagation of the extensional constraint still further via two efficient data structures. We demonstrate their effectiveness on both random and structured problems.

2 Background

The finite-domain *constraint satisfaction problem* (CSP) consists of: a finite set of variables, $\mathcal{X}$; for each variable $x \in \mathcal{X}$, a finite set $\mathcal{D}(x)$ of values (its domain); and a finite set $\mathcal{C}$ of constraints on the variables, where each constraint $c \in \mathcal{C}$ is defined over a subset of $\{x_i, \ldots, x_j\}$ of $\mathcal{X}$ (its *scope*, denoted scope(c)) by a subset of the Cartesian product $\mathcal{D}(x_i) \times \cdots \times \mathcal{D}(x_j)$ giving the set of allowed combinations of values. That is, c is a relation.

A *constrained optimisation problem* is a CSP with some objective function, which is to be optimised. A variable is *assigned* a value from its domain. A partial assignment is an assignment to one or more elements of $\mathcal{X}$. A solution is a partial assignment that includes all elements of $\mathcal{X}$ and satisfies all constraints. This paper focuses on the use of systematic search through the space of partial assignments to find such solutions.

An extensional or table constraint is simply the relational view of a constraint described above. That is, it lists explicitly the subset of the Cartesian product allowed, as in the example given in the introduction. This is as opposed to an intensional constraint, where the allowed assignments are computed via some algorithm. Typically, to propagate an extensional constraint we wish to enforce a property known as *generalised arc consistency* [10]. A constraint c is generalised arc consistent (GAC) if and only if for every variable x_i in scope(c) and every value v in D_i, there is at least one assignment to scope(c) that assigns v to x_i and satisfies c. Values for variables other than x_i participating in such assignments are known as the *support* for the assignment of v to x_i.

The GAC-Schema algorithm is commonly used to enforce GAC on an extensionally-represented constraint. It is shown in Algorithm 1. The algorithm was introduced by Bessiere and Régin [2] and the presentation here is taken from Lhomme and Régin [9]. The procedure SeekInferableValidSupport simply searches through the list $S_C(y, b)$ for a valid tuple, removing invalid tuples as it goes.

The procedure SeekValidSupport finds a new valid tuple supporting (y, b). The aim of this paper is to explore implementations of SeekValidSupport. This work is orthogonal to the GAC-Schema algorithm.

3 Example

Lhomme and Régin provide the following example. Consider the constraint with the following satisfying tuples:

Algorithm 1 GAC-Schema propagate

```
procedure GACSchemaPropagate(C: constraint, x: variable, a: value, deletionSet: list): Boolean
for each τ ∈ S_C(x, a):
    for each (z, c) ∈ τ: remove τ from S_C(z, c)
    for each (y, b) ∈ S(τ):
        remove (y, b) from S(τ)
        if b ∈ D(y):
            σ ← SeekInferableValidSupport(y, b)
            if σ ≠ nil:
                add (y, b) to S(σ)
            else:
                σ ← SeekValidSupport(C, y, b)
                if σ ≠ nil:
                    add (y, b) to S(σ)
                    for each x ∈ X(C):
                        add σ to S_C(x, σ[x])
                else:
                    remove b from D(y)
                    if D(y) = ∅: return false
                    add (y, b) to deletionSet
return true
```

$(x_1, x_2, x_3, x_4, x_5, x_6)$
(0, 0, 0, 0, 0, 0)
(0, 0, 0, 0, 1, 0)
(0, 0, 0, 0, 2, 0)
(0, 0, 0, 0, 3, 0)
(0, 0, 0, 0, 4, 0)
(0, 0, 0, 1, 0, 0)
(0, 0, 0, 1, 1, 0)
...
(0, 4, 4, 4, 4, 0)
(1, 1, 1, 1, 1, 1)
(2, 2, 2, 2, 2, 2)
(3, 3, 3, 3, 3, 3)
(4, 4, 4, 4, 4, 4)

Suppose that GAC has been established, with the tuple (0, 0, 0, 0, 0, 0) as the support for $(x_1, 0)$. If value $(x_6, 0)$ is pruned, the naive method of finding a new support for $(x_1, 0)$ would iterate through all 5^4 tuples up to (1, 1, 1, 1, 1, 1). There is scope for improvement here, by jumping over tuples which contain a value which has been pruned.

4 Next-Difference Lists

The most naive implementation of SeekValidSupport involves dividing the set of tuples into lists of supporting tuples for each variable x and value a, denoted tupleLists(x, a). SeekValidSupportSimple (algorithm 2) searches linearly through the list for a valid tuple. The index of the valid tuple is stored ($CS(x, a)$), and search is resumed from that point the next time SeekValidSupportSimple is called. This index is not backtracked, so it must be possible to restart in case the valid tuple(s) are before the stored index in the list.

The Next-Difference list is a simple improvement to this. Each item in the list is a record containing the tuple t, and a precomputed array of list indices called ND. $ND(x)$ is the index of the next tuple which

Algorithm 2 SeekValidSupportSimple

```
procedure SeekValidSupportSimple(x: variable, a: value): Tuple
i ← CS(x, a) { current support}
l ← tupleLists(x, a)
while ¬Valid(l(i)) and i ≤length(l):
    i ← i + 1
if Validl(i):
    CS(x, a) ← i
    return l(i)i
i ← 1 { Restart}
while ¬Valid(l(i)) and i < CS(x, a):
    i ← i + 1
if Validl(i):
    CS(x, a) ← i
    return l(i)
return nil
```

contains a *different* value for variable x. Therefore, if the current tuple contains value a for variable x, then $ND(x)$ has the index of the next tuple to contain $b \neq a$. If value a has been pruned, it is sound to skip to the next tuple not containing a.

To illustrate, consider Figure 1, which shows the Next-Difference list corresponding to a ternary constraint with scope $\langle x, y, z\rangle$, and allowed tuples: $\{\langle 0,0,0\rangle, \langle 0,0,1\rangle, \langle 1,1,0\rangle, \langle 1,1,1\rangle\}$. If value 0 is pruned from variable x, when searching for a support for $(z, 0)$, it is possible to jump from tuple 1 to tuple 3 in one step.

The procedure for searching this data structure is given in algorithm 3. The new algorithm can be used with one list containing all tuples, or with lists containing supporting tuples for each variable and value (x, a). The flag OneList, used on line 2 of algorithm 3 determines if one list is used. The lists are sorted in lexicographic order, with the leftmost value in the tuple as the most significant. Therefore it is likely that finding the leftmost invalid value would allow to jump forward the furthest, so we iterate from the left when checking the validity of the tuple. Towards the end of the list, *ND*(j) is likely to contain $\diamond$ indicating that there is no subsequent tuple with a different value for $t(j)$. $\diamond$ is considered greater than length(l).

Algorithm 3 behaves identically to the naive algorithm 2 if OneList is false and lines 12 and 22 (where it jumps forward) are replaced with $i \leftarrow i+1$. The only extra overhead is retrieving the new value of i from the *ND* array, and in the degenerate case where the first tuple examined is valid there is no extra overhead. Therefore there is no reason it should ever perform significantly worse than the naive algorithm, and it has potential to perform much better.

5 Tries

A trie is a tree data structure introduced by Fredkin [3] as an efficient means of storing and retrieving strings. The key idea is that strings with a common prefix share nodes and edges in the tree. Each node can have at most k children, where k is one more than the size of the alphabet, and each edge is labelled either with a character from the alphabet or a special terminating character. The root node has a child for each distinct first character, α, in the set of strings, and the edge connecting the two is labelled with α. Each internal node n has a child for each string that has a prefix corresponding to the characters on the edges in the path from the root to n, read in order. To illustrate, Figure 2 shows the trie containing the strings 'a', 'an', 'and', 'as', 'ask', 'asp', 'be', and 'bet'. Searching for a string in a trie (the main operation useful for supporting a table constraint), is O(1).

We can view the tuples in the extensional representation of a constraint as strings and insert them into

Algorithm 3 SeekValidSupportNextDifference

```
procedure SeekValidSupportNextDifference(x: variable, a: value): Tuple
i ← CS(x, a) { current support}
if OneList: l is the global tuple list
else: l ← tupleLists(x, a)
while i ≤length(l):
    j ← 1 { Index into tuples from left}
    while j ≤ r and l(i).t(j) ∈ D_j and var(j) = x ⇒ l(i).t(j) = a:
        j ← j + 1
    if j = r + 1:
        CS(x, a) ← i
        return l(i)
    else:
        i ← l(i).ND(j) { Jump to the next tuple with jth value different }
i ← 1 { Restart}
while i < CS(x, a):
    j ← 1 { Index into tuples from left}
    while j ≤ r and l(i).t(j) ∈ D_j and var(j) = x ⇒ l(i).t(j) = a:
        j ← j + 1
    if j = r + 1:
        CS(x, a) ← i
        return l(i)
    else:
        i ← l(i).ND(j) { Jump to the next tuple with jth value different }
return nil
```

a trie. The branches of the trie will have a uniform length, and each level of the trie will correspond to a particular variable in the scope of the constraint represented. Testing whether a particular tuple satisfies the constraint is then cheap. For enforcing GAC, however, we wish to test whether a particular variable, value pair has support. If the variable is the first in the constraint's scope, this remains cheap.

To illustrate, consider Figure 3, which shows the trie corresponding to a ternary constraint with scope $\langle x, y, z\rangle$, and allowed tuples: $\{\langle 0,0,0\rangle, \langle 0,0,1\rangle, \langle 1,1,0\rangle, \langle 1,1,1\rangle\}$. To establish support for $x = 0$ we follow the arc corresponding to 0 from the root and find a path to a leaf node where the value labelling each edge remains in the domain of the corresponding variable. If, say, the value 0 has been removed from the domain of z, the support established for $x = 0$ is as shown in the figure ($\langle 0,0,1\rangle$).

If, however, the variable is the last in the scope, search for support can be very expensive. Consider, for example, seeking support for a value of z in the example of Figure 3. In this case, we might need to explore a large proportion of the leaves of the trie. Therefore, we trade space for time and use one trie per element e of the scope. In each trie we arrange the levels so that e is represented at the first level. The two further tries for the example of Figure 3 are given in Figure 4. It is now cheap to establish support for any variable in the scope.

Tries can also be used to re-establish support efficiently when it is lost. Assume that, having, established support for a value v in the domain of some variable x, that propagation has removed one or more values from the domains of the other variables in the scope. From the leaf corresponding to the tuple supporting $x = k$, we ascend to find the highest level in the trie whose corresponding variable has lost a value in this tuple. From the parent node of this edge, we begin a search for new support from the next child to the right of the edge whose value was lost.

The tries are implemented as two-dimensional arrays, indexed firstly by trie level ($1 \ldots r$ from root to leaves). In the other dimension, each entry corresponds to one node in the trie. Blocks of nodes with the same parent are separated by a special delim value. Each node has a value (accessed by .value), an index to its leftmost child (.child) and an index to its parent (.parent). This is illustrated in figure 5, for a trie

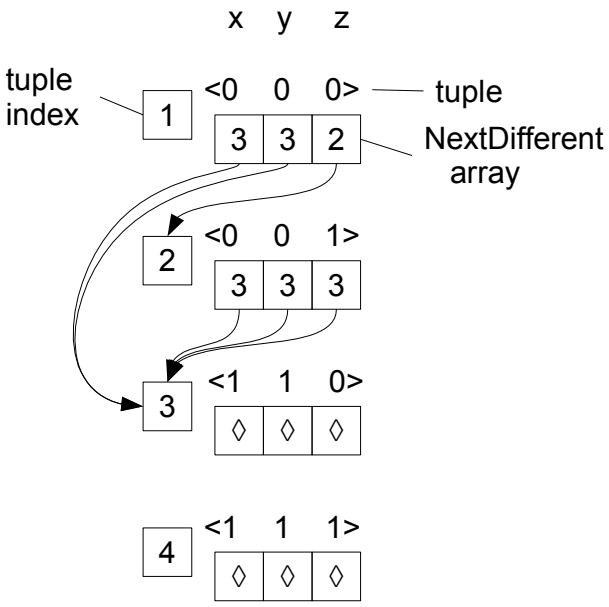

Figure 1: Next-Difference list example

containing tuples $\langle 1,2,2\rangle, \langle 1,3,3\rangle, \langle 2,1,2\rangle, \langle 2,3,3\rangle, \langle 3,1,3\rangle, \langle 3,2,3\rangle$.

The procedure to search the trie is given in algorithm 4. It searches the appropriate trie from the point it left off in the previous call, starting at a leaf. When it reaches an end condition, it restarts and sets the searchUpTo flag to true. Then it searches up to the indices stored in the minLevel array. There is one index stored for each level of the trie, and the algorithm stops if it reaches the minLevel whatever level of the trie it is on. If the algorithm reaches an end condition again, then there are no valid tuples to support x, a so nil is returned.

The algorithm starts from a leaf, where it finished the previous time it was invoked. It ascends the trie to find the highest node where the value is invalid. (If this does not exist, the previous support is still valid.) From this point, search is started.

The search loop is divided into two parts. The first part checks for end conditions, such as reaching the top level of the trie, or reaching the right end of one level of the trie. These might stop the search, or restart it. Also, the minLevel array is populated here, if the current index is smaller than the one stored in the array.

The second part of the search loop navigates the trie. This is divided into three parts, one of which is executed. The first moves to the parent nodes' successor if we have reached the end of the current block. The second part checks if the value of the current node is valid. If it is, then we can move down one level. (If we are on the bottom level, we have found a valid tuple, so return it.) If neither condition holds, we move to the next node on the current level.

Algorithm 4 seekNextSupportTrie

procedure SeekValidSupportTrie(x: variable, a: value): Tuple
$i \leftarrow CS(x, a)$
trie$\leftarrow$ tries(x) { Select the appropriate trie }
$hl \leftarrow r + 1$ {highest level with invalid value}
for l in $r \ldots 1$: {l: trie level}
    if trie[l][i].value $\notin D(x_l)$:
        $hl \leftarrow l$ and $hi \leftarrow i$
    $i \leftarrow$trie[l][i].parent
if $hl = r + 1$: **return** reconstructTuple(i) {support still valid}
$i \leftarrow hi$
$l \leftarrow hl$
searchUpTo$\leftarrow$*false*
$\forall l$:minLevel[l]$\leftarrow +\infty$
while true: {search for new support}
    if searchUpTo **and** [$l = 0$ **or** $i >$minLevel[l] **or** $i =$length(trie[l])]:
        return nil
    if $l = 0$ **or** $i =$length(trie[l]):
        searchUpTo$\leftarrow$*true* and $l \leftarrow 1$
        $i \leftarrow$ lowest index corresponding to value a
    if $\neg$searchUpTo **and** minLevel[l]$> i$:
        minLevel[l]$\leftarrow i$
    {navigate the tree}
    if trie[l][i]=delim:
        $i \leftarrow$trie[l][$i - 1$].parent+1
        $l \leftarrow l - 1$
    else if trie[i][l].value $\in D(x_l)$:
        if $l = r$:
            $CS(x, a) \leftarrow i$
            return reconstructTuple(i)
        $i \leftarrow$trie[l][i].child {the first child}
        $l \leftarrow l + 1$
    else:
        $i \leftarrow i + 1$

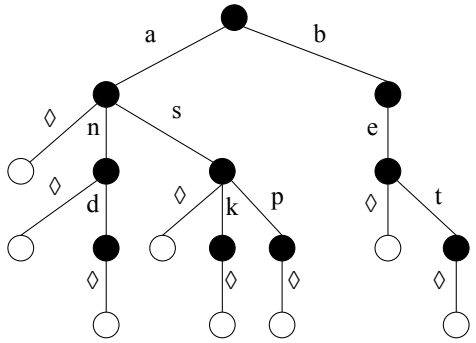

Figure 2: Trie containing words

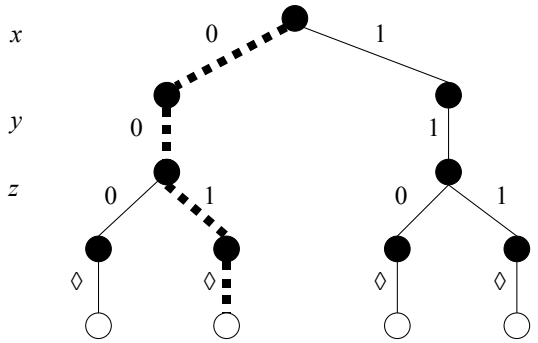

Figure 3: Trie containing tuples

6 Experiments

Implementation The algorithms described above were implemented in Java 1.5 and embedded in GAC-Schema (algorithm 1). This is called from a simple queue embedded in a search procedure with static variable and value orderings. Since both constraint scopes and microstructure are generated randomly with uniform distribution, the actual variable and value orderings are irrelevant. Search is performed until the first solution is found, or the search space has been exhausted.

To obtain the timings, a microsecond timer was used for each call to SeekNextSupport, and these were added together for all calls during search, for all constraints in the problem instance. Timings for 20 or 50 instances were added together giving a total time in microseconds spent in SeekNextSupport. The machine used was a Pentium 4 3.06GHz with hyperthreading switched off, and 1GB of RAM.

Problem instances CSP instances were generated with the parameters in figure 6. For each line in the table, l, r, d and n are fixed and a range is given for e. All integers in the range are used, each making one unique tuple $\langle l, r, d, n, e \rangle$. For each unique tuple, a suite of instances were generated (50 if the density is 0.2, and 20 otherwise). This is a total of 4700 instances for density 0.2, 4220 instances for density 0.5 and

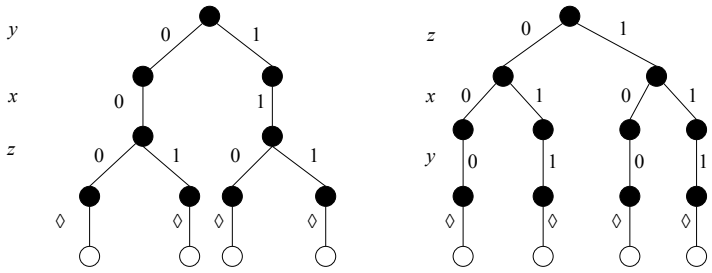

Figure 4: Tries for the two other relevant orderings of x, y, z

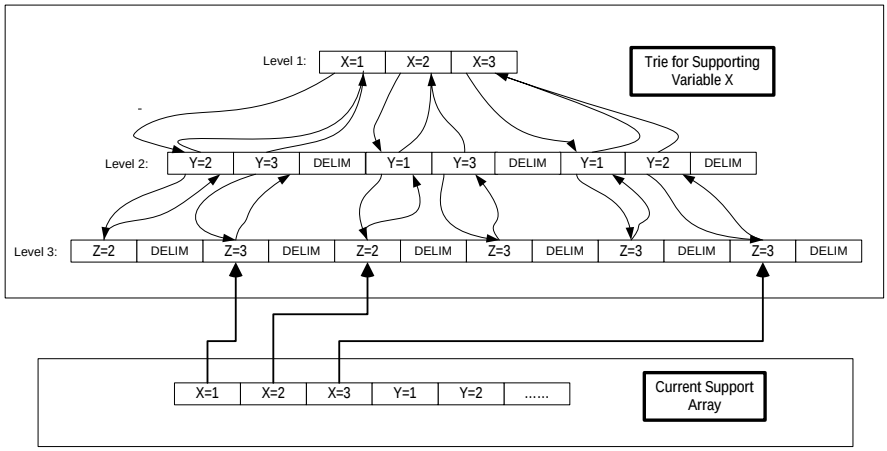

Figure 5: Illustration of the trie data structure for finding support for variable X

5060 instances for density 0.8. Overall 13980 instances were generated.

Density (looseness) is the proportion of satisfying tuples in each constraint, chosen with uniform probability. For each constraint, its r variables are chosen with uniform distribution from the n variables in the problem. The constraint hypergraph is not necessarily connected.

The range for e was chosen to cross the phase transition. At the lowest value, all (50 or 20) instances in the suite are satisfiable. The highest value was chosen so that more than 75% of the suite are unsatisfiable.

Solution For each set of parameters, the suite of instances is solved using a static arbitrary variable and value ordering, and the times used in the following graphs are the sum of the times over the suite. Both the total time to solve, and the time spent in seekNextSupport are recorded, in microseconds.

The first data set we present shows that there seems to be at best a marginal improvement storing backtrackable pointers in memory. This is shown in Figure 7. In general we get an improvement in time for seeking support by backtracking pointers, but very rarely more than 25%, a minor improvement compared to those we will see in a moment. However, when integrated into search, the additional overhead of storing these pointers is often not repaid. Here, we see some instances speeding up and some slowing down when pointers are stored in backtrackable memory. Because of this (at best) marginal improvement, for the rest

Density (looseness) l	Arity r	Domain size d	Number of vars n	Number of cons e
0.2	5	2	25	$1 \ldots 9$
	5	3	13	$1 \ldots 10$
	5	4	8	$1 \ldots 8$
	5	5	7	$1 \ldots 8$
	7	2	24	$1 \ldots 11$
	7	3	12	$1 \ldots 10$
	7	4	7	$1 \ldots 7$
	9	2	23	$1 \ldots 11$
	9	3	11	$1 \ldots 9$
	11	2	22	$1 \ldots 11$
0.5	5	2	25	$1 \ldots 24$
	5	3	13	$1 \ldots 22$
	5	4	8	$1 \ldots 18$
	5	5	7	$1 \ldots 18$
	7	2	24	$1 \ldots 26$
	7	3	12	$1 \ldots 20$
	7	4	7	$1 \ldots 15$
	9	2	23	$1 \ldots 24$
	9	3	11	$1 \ldots 20$
	11	2	22	$1 \ldots 24$
0.8	5	2	23	$30 \ldots 76$
	5	3	11	$30 \ldots 58$
	5	4	6	$30 \ldots 38$
	7	2	22	$30 \ldots 70$
	7	3	10	$30 \ldots 52$
	9	2	21	$30 \ldots 71$
	9	3	9	$30 \ldots 52$
	11	2	20	$30 \ldots 68$

Figure 6: Table of CSP parameters

of the paper we compare new techniques to the simple method without backtrackable pointers being stored.

The next data set, in Figure 8, shows that the use of Tries can improve time in SeekNextSupport by more than 20 times, and overall run time by more than 5 times. Improvement seems to be greatest in our experiments with density 0.2 and 0.5. There are still data points where Simple beats Tries, but this is only dramatic at very small run times and so where a significant factor slowdown occurs, it corresponds to little in absolute run time. While many data points show very similar performance using the two techniques, on the ensemble of data as a whole, Tries are well worthwhile since they can lead to very dramatic improvements on large run times.

The next data set shows (in Figure 9) that the use of Next-Difference lists can be highly successful, but unfortunately memory problems seem to cause this technique to be untenable overall as it causes slowdown and even failure to complete in some cases. It would be interesting to see if some variation of this technique can be used to ameliorate this problem. The use of a single list instead of multiple lists does seem to reduce memory problems, but we still see, in Figure 10, that it can be significantly slower on some instances.

Next we report results using Regin and Lhomme's hologram tuples in Figure 11. Significant factor

improvements in run time are obtained on many instances. There is no pathological behaviour as we saw with Next-Difference lists, but there is a consistent slight slowdown on the density 0.8 instances. To clarify the difference between this data structure and Tries, we also show a plot of the two techniques compared directly in Figure 12. This shows clearly that Tries are almost always faster in both SeekNextSupport and overall runtime, clustering in the range from 1.5 to 2.5 times as fast in SeekNextSupport. The speedup is much more consistent than in the previous plots, suggesting that similar factors allow both techniques to run faster than Simple, but that Tries are able to do so more effectively. While we get consistent speedups over hologram tuples, the speedups are small enough that we cannot rule out that a different implementation might reverse the situation. We can only say that we certainly have tried to implement both techniques to the best of our ability.

The results above show that Tries is the best performing of the techniques we have studied on this data set. We now examine its behaviour in more detail. Results in raw run-time and in comparison with Simple are shown in Figure 13 for density 0.2, Figure 14 for density 0.5, and Figure 15 for density 0.8. For density 0.8 we see that all plots show little variation from the ratio 1, i.e. that Tries are never much better than Simple in this case, nor are they ever much worse. The data for the other densities is more interesting. At both 0.2 and 0.5, careful examination (for which the colour plots are very helpful if available to the reader) shows two interesting trends in both plots. First, if we fix the arity and increase domain size, then Tries become increasingly good relative to Simple. Second, if we fix the domain size and increase the arity, then again Tries become increasingly good. This trend is without exception in terms of the of each line, with just one or two exceptions at individual points in the plots. This behaviour can be summarised by saying that as the size of the tables increases, the benefits of using Tries increases too.

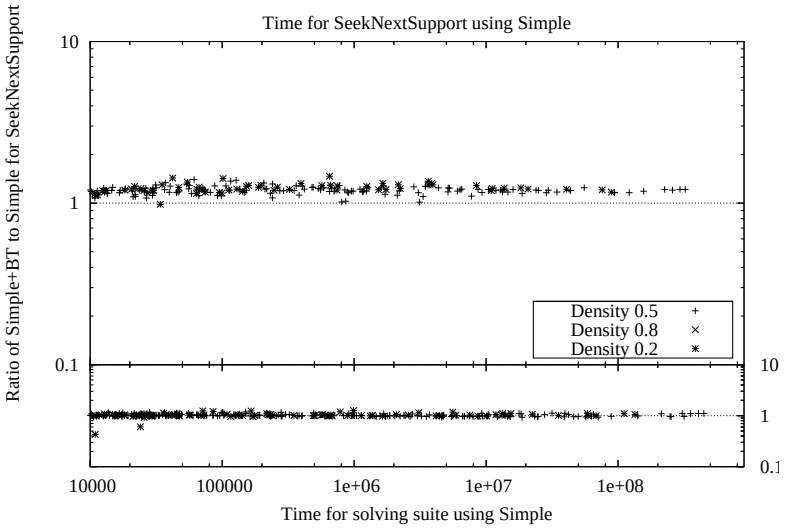

Figure 7: Comparison of performance of the Simple algorithm with and without storing pointers and restoring them on backtracking. Two scatterplots are shown in this figure. The upper plot is labelled to the left and above the graph, while the lower plot is labelled to the right and beneath the plot. Note that all scales are logarithmic, and that the scale of the x-axis is the same in both plots. We first discuss the upper (larger) plot. On the x-axis we show the run time (in microseconds) used in SeekNextSupport. Each point represents the sum of all the times in a suite of 20 or 50 instances. We omit those datasets which required less than 0.01s for the suite, but all other parameter sets are included. On the y-axis, we show the ratio of this time to the time used for SeekNextSupport when pointers are backtracked in the Simple algorithm. In this case, we see that most data sets use less time in SeekNextSupport with backtracked pointers, although the improvement is marginal. The lower plot is similar, but time measured is now the run time for solving the suite, instead of simply time inside SeekNextSupport. Thus this takes account of any overheads (such as storing and restoring pointers.) Here, we see that most points are very close to 1, with some above and some below that line, with almost none far from the line.

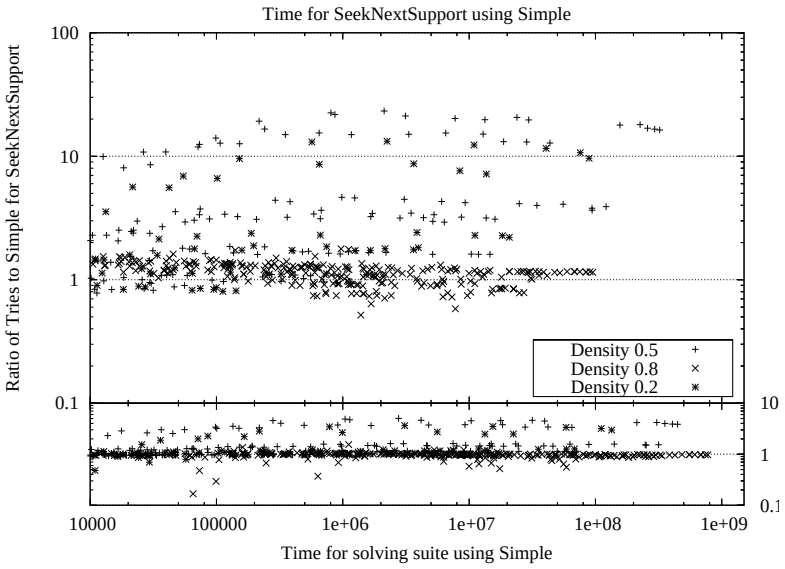

Figure 8: Comparison of performance of using Tries with use of the Simple algorithm. Two scatterplots are shown in this figure: for more details of the plots, please refer to the caption of Figure 7. In this case, we see that most data sets use less time in SeekNextSupport with Tries. In many cases the improvement is very significant, sometimes more than 20 times. The biggest improvements appear at densities 0.2 and 0.5. There are a number of cases where Simple is faster, mainly at density 0.8, although these reduce in frequency as overall runtime increases: to a limited extent this suggest that Tries improves as instances becomes harder. Also, note that Simple is never even twice as fast as Tries. The lower plot shows run time for solving the suite, instead of simply time inside SeekNextSupport. The use of tries can lead to a several-fold improvement in run time overall. Simple can be significantly faster, but only on data points with small run times.

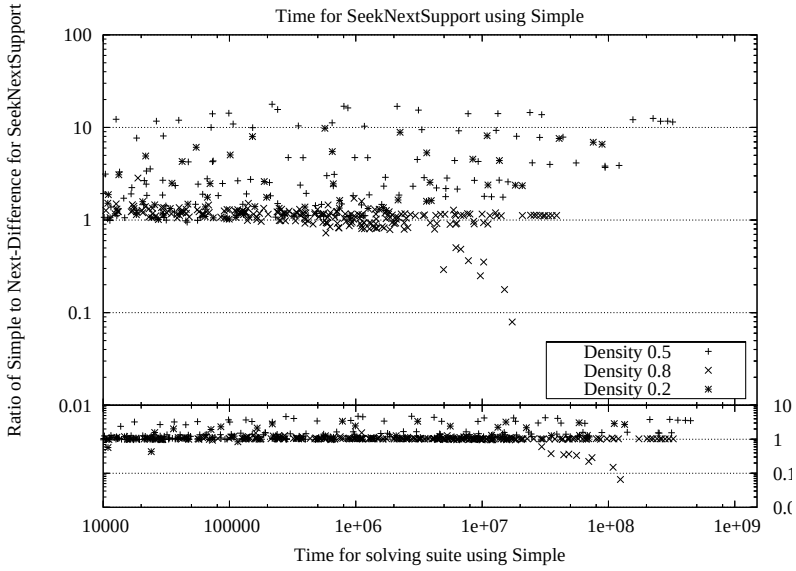

Figure 9: Comparison of performance of using Next-Difference Lists with use of the Simple algorithm. Two scatterplots are shown in this figure: for more details of the plots, please refer to the caption of Figure 7. As with Tries, we can obtain very significant speedups over Simple. Unfortunately, there can also be very significant slowdowns. We believe this is associated with memory usage becoming too large. In fact, memory problems led to many instances being unable to complete, these being necessarily omitted from the plot.

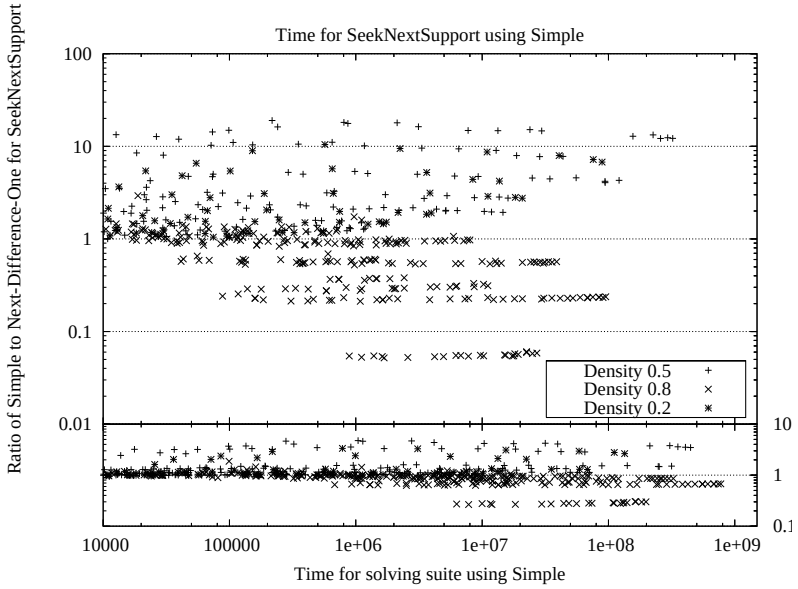

Figure 10: Comparison of performance of using Next-Difference List (using only one List) with use of the Simple algorithm. Two scatterplots are shown in this figure: for more details of the plots, please refer to the caption of Figure 7. While we did not have problems with data sets failing to complete due to memory problems, we see that poor performance results in many instances at density 0.8.

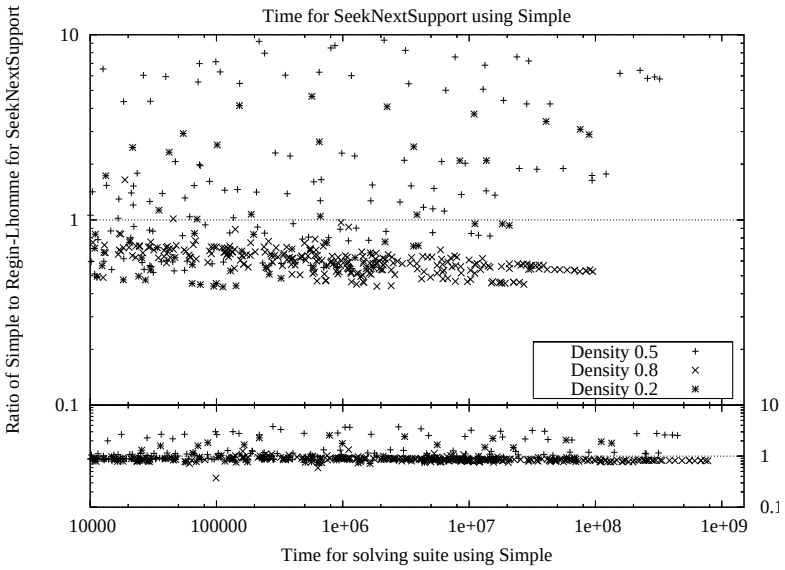

Figure 11: Comparison of performance of using hologram tuples with the use of the Simple algorithm. Two scatterplots are shown in this figure: for more details of the plots, please refer to the caption of Figure 7. We see that significant speedups can be obtained (both in SeekNextSupport and overall), but that on the density 0.8 instances a consistent slight slowdown happens.

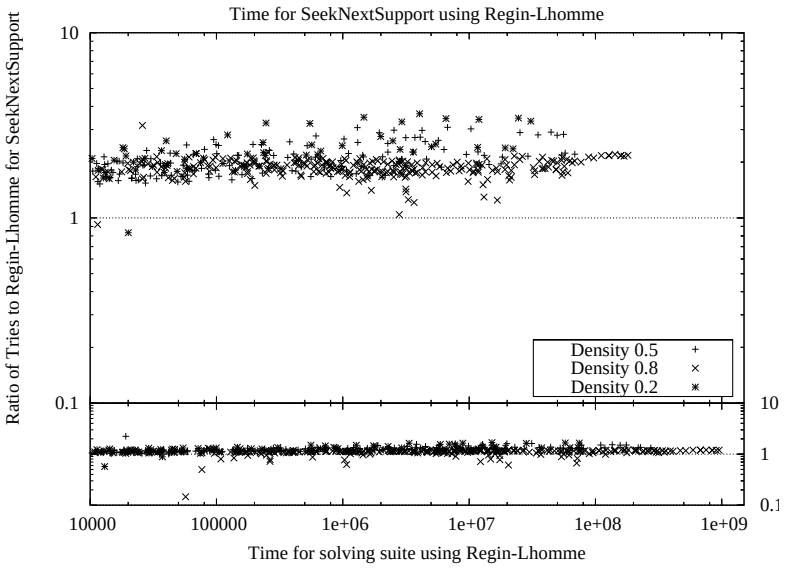

Figure 12: Comparison of performance of using Tries and using hologram tuples. Plots are in the same style as Figure 7, but we plot time using hologram tuples on the x-axis, and ratio of that technique to using Tries on the y-axis. We see that Tries are almost always faster, clustering at about twice as fast in SeekNextSupport.

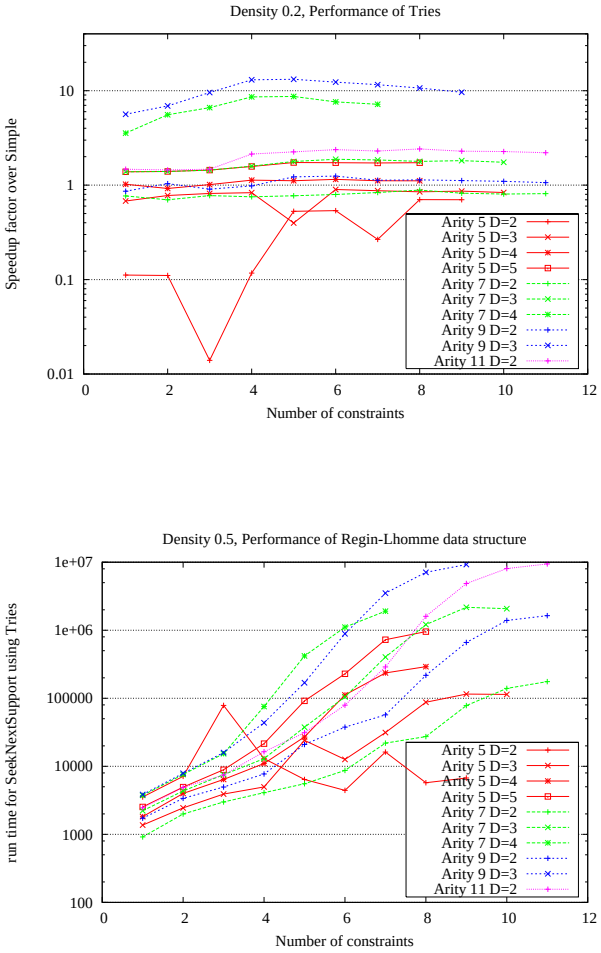

Figure 13: Performance of Tries at density 0.2, with differing arities and domain sizes. The x-axis of both plots shows the number of constraints, while in both plots lines join points with the same arity and domain size. The top plot shows the performance improvement of using Tries over Simple for SeekNextSupport. The bottom shows the absolute runtime (over the set of instances in microseconds) for SeekNextSupport using Tries.

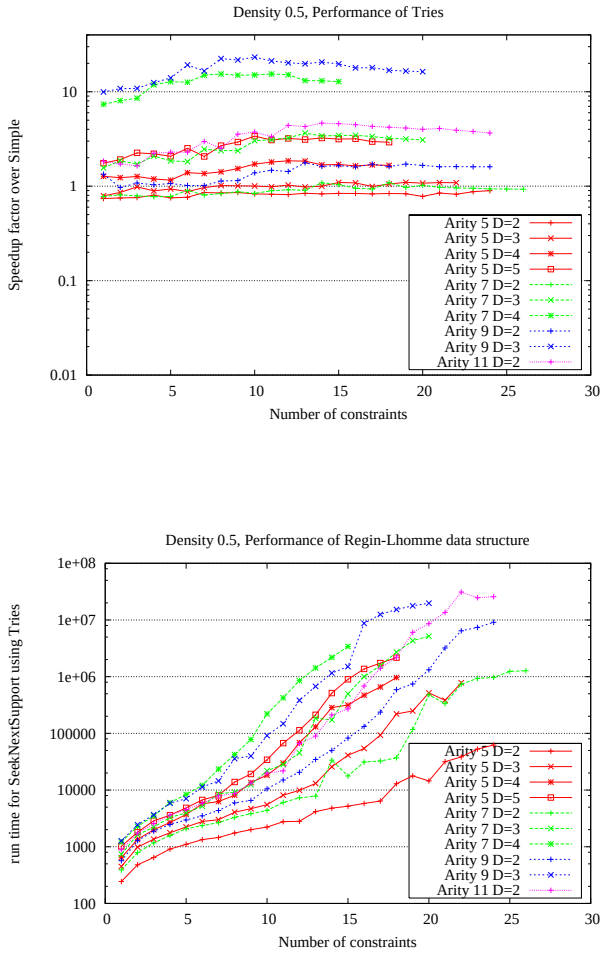

Figure 14: Performance of Tries at density 0.5, with differing arities and domain sizes. The x-axis of both plots shows the number of constraints, while in both plots lines join points with the same arity and domain size. The top plot shows the performance improvement of using Tries over Simple for SeekNextSupport. The bottom shows the absolute runtime (over the set of instances in microseconds) for SeekNextSupport using Tries. Note that each arity at a single domain size uses the same point marking, while each domain size for a fixed arity uses the same line marking (and colour if available).

7 Conclusions

This paper has described the use of Next-Difference lists and tries in maintaining generalised arc consistency on table/extensional constraints. Our experiments have demonstrated their utility for this application. It was observed in the experiments that one method (Next-Difference lists) started to perform poorly because of memory problems. Where t is the number of tuples, the space complexity of hologram tuples is $O(tr)$ (or more precisely, $4tr$ machine words). For Next-Difference lists with one list, it is $O(tr)$ ($2tr$ words). For Next-Difference lists with one list per variable, it is $O(tr^2)$ ($tr + tr^2$) which causes problems on some of the larger instances. Tries are also $O(tr^2)$ but because the tuples are compressed together at the top of the trie, actual memory usage is much less than the worst case.

At the time of writing, we have become aware of Lecoutre and Szymanek's work on table constraints [8]. Comparing their approach to our own is an important item of future work.

Acknowledgments

This work was in part supported by an EPSRC Network grant no GR/S86037/01. Ian Miguel is supported by a UK Royal Academy of Engineering/EPSRC Research Fellowship. Peter Nightingale is supported by an EPSRC doctoral training grant. We thank Chris Jefferson for helpful discussions on data structures and the table constraint in general. We thank Christophe Lecoutre and Radoslaw Szymonek for sending us a preprint of their CP 06 paper.

References

[1] P. Baptiste, C. Le Pape. Constraint Propagation and Decomposition Techniques for Highly Disjunctive and Highly Cumulative Project Scheduling Problems. *Constraints* 5(1/2), 119-139, 2000.

[2] C. Bessiere, J-C. Regin. Arc Consistency for General Constraint Networks: Preliminary Results. *IJCAI*, 398-404, 1997.

[3] E. Fredkin. Trie Memory. *Communications of the ACM* 3(9), 490-499, 1960.

[4] A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh. Propagation Algorithms for Lexicographic Ordering Constraints. *Artificial Intelligence* 170(10), 803-834, 2006.

[5] A.M. Frisch, I. Miguel, T. Walsh. Symmetry and Implied Constraints in the Steel Mill Slab Design Probem. *CP'01 Workshop on Modelling and Problem Formulation*, 8-15, 2001.

[6] I.P. Gent, C. Jefferson, I. Miguel. Minion: A Fast, Scalable Constraint Solver. *17th European Conference on Artificial Intelligence*, 2006.

[7] ILOG. *ILOG Solver 6.3 User Manual*, ILOG, S.A. Gentilly, France, 2006.

[8] C. Lecoutre, R. Szymanek. Generalized Arc Consistency for Positive Table Constraints *12th International Conference on Principles and Practice of Constraint Programming*, 2006.

[9] O. Lhomme, J-C. Regin. A Fast Arc Consistency Algorithm for n-ary Constraints. *AAAI*, 405-410, 2005.

[10] A.K. Mackworth. On Reading Sketch Maps. *IJCAI*, 598-606, 1977.

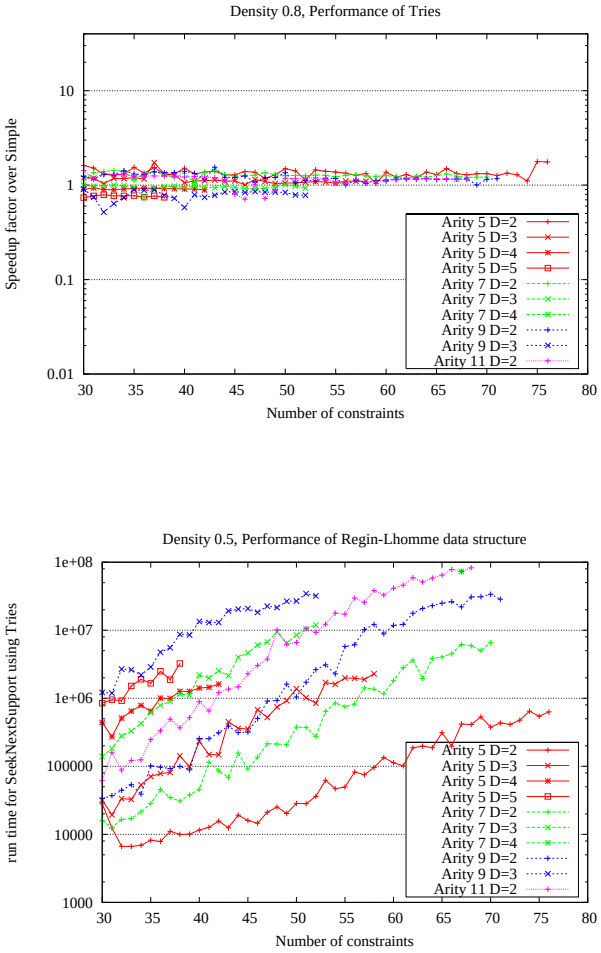

Figure 15: Performance of Tries at density 0.8, with differing arities and domain sizes. The x-axis of both plots shows the number of constraints, while in both plots lines join points with the same arity and domain size. The top plot shows the performance improvement of using Tries over Simple for SeekNextSupport. The bottom shows the absolute runtime (over the set of instances in microseconds) for SeekNextSupport using Tries.

[11] J-C. Regin. A Filtering Algorithm for Constraints of Difference in CSPs. *AAAI*, 362-367, 1994.

[12] B.M. Smith, K. Stergiou, T. Walsh. Using Auxiliary Variables and Implied Constraints to Model Non-binary Problems. *AAAI*, 182-187, 2000.

[13] M. Wallace. Practical Applications of Constraint Programming. *Constraints* 1(1/2) 139-168, 1996.