Data Structures for Generalised Arc Consistency for Extensional Constraints*

Ian P. Gent,[1] **Chris Jefferson,**[2] **Ian Miguel,**[1] and **Peter Nightingale**[1]
[1]School of Computer Science, University of St Andrews, Fife, KY16 9SX, United Kingdom
{ipg, ianm, pn}@cs.st-and.ac.uk
[2]Oxford University Computing Laboratory, University of Oxford, Oxford OX1 3QD, United Kingdom
chris.jefferson@comlab.ox.ac.uk

Abstract

Extensional (table) constraints are an important tool for attacking combinatorial problems with constraint programming. Recently there has been renewed interest in fast propagation algorithms for these constraints. We describe the use of two alternative data structures for maintaining generalised arc consistency on 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, both in GAC-schema, and in the watched-literal table constraint in Minion.

Introduction

Constraint programming is a successful technology for solving a wide variety of combinatorial problems from industry and academia. Examples include scheduling, industrial design, and combinatorial mathematics, to name but a few examples (Wallace 1996). A backtracking 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. Various constraint solving systems are available, e.g. ILOG Solver, Eclipse `http://eclipse-clp.org`, Gecode `http://gecode.org`, and Minion `http://minion.sourceforge.net`. All offer a library of constraints with which to model problems.

The extensional or 'table' constraint is an important part of a constraint library. It allows us to simply 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 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. $\{\langle$Bill, Bert$\rangle$, $\langle$Bill, Tom$\rangle$, $\langle$Bert, Tom$\rangle\}$, This represents that Bill likes Bert and Tom, Bert likes Tom, and Tom likes noone. Expressing the same facts using combinations of other constraints, such as inequalities and implications, can be an awkward task.

The ability to establish generalised arc consistency (GAC) on table constraints can be a powerful tool, allowing great reductions in search over other formulations. It is essential to be able to propagate this type of constraint as efficiently as possible. While there has been extensive research into algorithms of GAC, a very recent trend has been to investigate data structures which enable satisfying tuples to be found quickly. There is great potential in this area. The naive approach, which is simply to traverse the list of satisfying tuples in search of 'support' for a given assignment, is potentially expensive. Recently, Lhomme and Régin (2005) introduced the *Hologram* data structure for tuples to improve this process, and Lecoutre and Szymanek (2006) have used binary search. In this paper, we continue to tap this potential by introducing two new ways of improving the propagation of the extensional constraint via two efficient data structures. We show their effectiveness on both random and structured problems in two implementations: our own of GAC-schema, and when integrated into the constraint solver Minion.

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. A solution is a partial assignment that includes all elements of $\mathcal{X}$ and satisfies all constraints.

An extensional or table constraint is simply the relational view of a constraint described above. 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

*Supported in part by EPSRC Network grant GR/S86037. Ian Miguel is supported by a UK Royal Academy of Engineering/EPSRC Research Fellowship. Peter Nightingale is supported by an EPSRC doctoral training grant. Chris Jefferson is supported by EPSRC Grant EP/D032636. We thank Christophe Lecoutre, Radoslaw Szymonek and AAAI reviewers. We thank the School of Computer Science at St Andrews for donation of machine time.

as *generalised arc consistency* (Mackworth 1977). 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. The algorithm was introduced by Bessière and Régin (1997) and we use the version from Lhomme and Régin (2005). Unfortunately we omit pseudocode for space reasons. The focus of this paper is on the implementation of the procedure SeekValidSupport, which finds a new valid tuple supporting (y, b) when the current support for $y = b$ is no longer valid.

Lhomme and Régin (2005) argued that SeekValidSupport should search only allowed tuples, while only considering tuples containing values which are valid (in the current domain of their variable.) No way is known to achieve this goal perfectly. Lhomme and Régin's provided one way of approximating this: we now introduce two new methods.

Next Difference Lists

The naive implementation of SeekValidSupport (as proposed in the original paper on GAC-schema) involves dividing the set of tuples into lists of supporting tuples for each variable x and value a, denoted tupleLists(x, a).[1] SeekValidSupportSimple searches linearly through the list for a valid tuple. The index of the valid tuple is stored ($CS(x, a)$

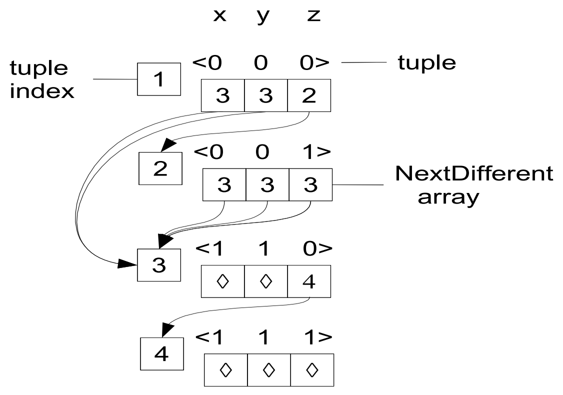

Figure 1: Next-Difference list example

where CS is Current Support), 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.[2] The Next-Difference list is a simple improvement to the simple approach. 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 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 1. 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 1 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 ◇ indicating that there is no subsequent tuple with a different value for $t(j)$. ◇ is considered greater than length(l). Algorithm 1 behaves identically to the naive algorithm 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.

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

Algorithm 1: SeekValidSupportNextDifference

[1]An alternative naive method is to generate possible tuples from variables' current domains, until one is found that is allowed by the constraint.

[2]Alternatively, we can store the pointers and restore them on backtracking, avoiding the restart. In our experiments we have not seen results differing by more than 25% when doing this.

Tries

A trie is a tree data structure introduced by Fredkin (1960) 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. Searching for a string in a trie (the main operation useful for supporting a table constraint), is $O(d)$ if d is the length of the string.

We can view the tuples in the extensional representation of a constraint as strings and insert them into 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 2, 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. This is a search process for which we present an algorithm later in this section. 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. 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 this example are given in Figure 3.

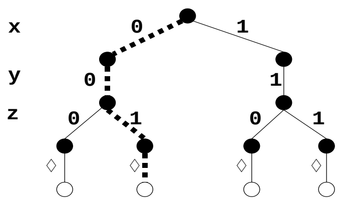

Figure 2: Trie containing tuples

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

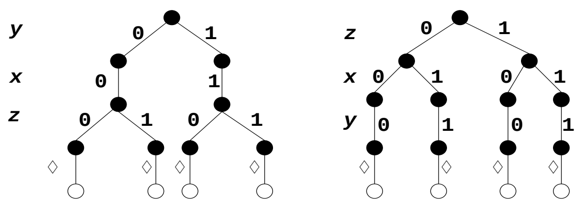

Figure 3: Tries for the two other relevant orderings of x, y, z

to the tuple supporting $x = v$, 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.

```
procedure SeekValidSupportTrie($x$: variable, $a$: value): Tuple
$i \leftarrow CS(x, a)$
trie← 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←false
$\forall l$ :minLevel[$l$]← $+\infty$
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←true and $l \leftarrow 1$
        $i \leftarrow$ lowest index corresponding to value $a$
    if ¬searchUpTo and minLevel[$l$]$> i$:
        minLevel[$l$]← $i$
    {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:
        $i \leftarrow i + 1$
```

Algorithm 2: seekValidSupportTrie

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). The procedure to search the trie is given in algorithm 2. It searches the appropriate trie from the point it left off in the previous call, starting at a leaf. That is, $CS(x, a)$ points to a leaf node representing current support. 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.

We report below that Next-Difference lists can perform poorly because of memory problems. Where t is the number of tuples, the space complexity of Hologram is $O(tr)$ (specifically $4tr$ machine words). For Next-Difference lists with one list, it is $O(tr)$ (specifically $2tr$ words). For Next-Difference lists with one list per variable, it is $O(tr^2)$ (specifically $tr + tr^2$) 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, typical memory usage is much less than the worst case.

Experiments with GAC-schema

The algorithms described above were implemented in Java 1.5 and embedded in GAC-schema. This is called from a simple queue embedded in a search procedure with static variable and value orderings. We used static orderings for simplicity, but we see no reason why similar results would not be obtained with dynamic variable ordering: speedups come from improving propagation time, not any change in the search tree. To obtain the timings, a microsecond timer was used for each call to SeekValidSupport. The machine used was a Pentium 4 3GHz with hyperthreading switched off, and 2GB of RAM.

Random CSP instances were generated with a variety of parameters. We used three values of looseness (the proportion of satisfying tuples): 0.2, 0.5, and 0.8 and in each case the tuples are chosen with uniform probability. We used constraints of arity 5, 7, 9, and 11. We also varied domain sizes, with a range of 2…5 for arity 5, decreasing to only domain size 2 for arity 11. To keep search manageable, we used no more than 25 variables for domain size 2, decreasing this with increasing domain size. For each combination, we used a range of integer values for the number of edges e, covering the phase transition (from 100% satisfiable to at least 75% unsatisfiable.) For each constraint, its variables are chosen with uniform distribution from the variables in the problem. The constraint hypergraph is not necessarily connected (again for simplicity of implementation.) For each combination of parameters, a suite of instances were generated (50 if the looseness is 0.2, and 20 otherwise).

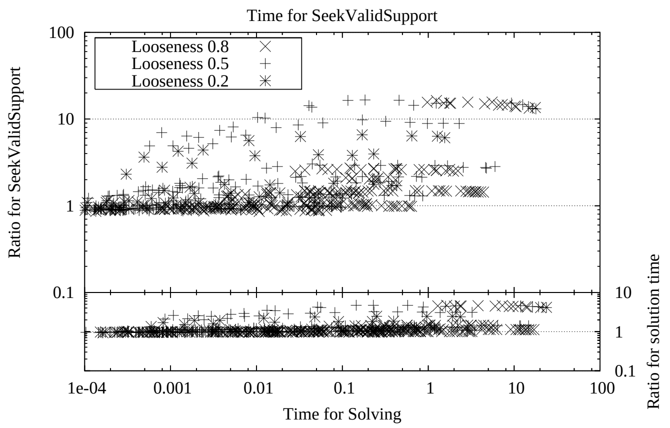

Figure 4: Comparison of performance of using Tries with use of the Simple algorithm. See text for explanation of this figure and similar ones to follow

We compare results using Tries and Simple in Figure 4. The format of the figure needs some explanation. 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 seconds) used in SeekValidSupportSimple. Each point represents the mean of the times in a suite of instances. On the y-axis, we show the ratio of this time to the time used for SeekValidSupportTries. The lower plot is similar, but time measured is now the mean run time for solving the suite, thus including all overheads.

Figure 4 shows that the use of Tries can improve time in SeekValidSupport by more than 10 times, and overall run time by more than 5 times. There are still data points where Simple beats Tries, but in all cases the difference is marginal. 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.

We found that Next-Difference lists (using multiple lists) can be highly successful, but memory usage cause this technique to be untenable overall. Memory problems often cause slowdown or even failure to complete. The use of a single list instead of multiple lists solves memory problems. We see (in Figure 5) that we achieve very similar performance to Tries with this algorithm. Tries never take more than 147%,

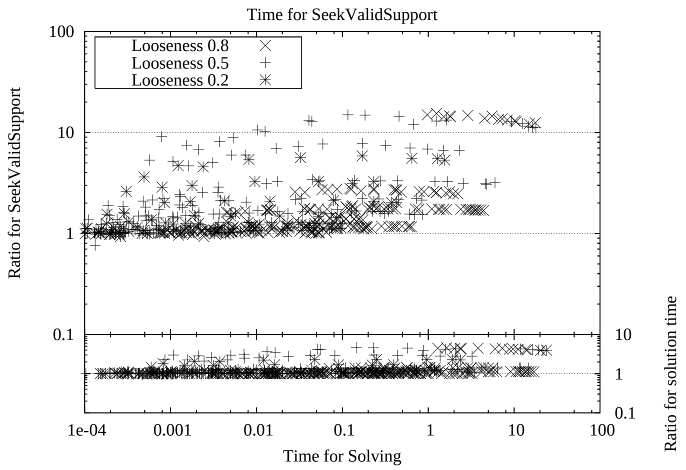

Figure 5: Comparison of using Next-Difference Lists (with one List) with use of the Simple algorithm.

or less than 75%, of the time that Next-Difference lists take in SeekValidSupport. The figures are usually very close.

Next we report results using Lhomme and Régin's Hologram data structure. Significant factor improvements in run time are obtained on many instances over Simple, but there is a consistent slight slowdown on the looseness 0.8 instances. Figure 6 shows a a direct comparison between this data structure and Tries. This shows clearly that Tries are always faster in both SeekValidSupport and overall runtime, clustering in the range from 1.5 to 2.5 times as fast in SeekValidSupport. As run time increases, Tries seem to become increasingly more effective. 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. Speedup in overall time is slight, but consistent.

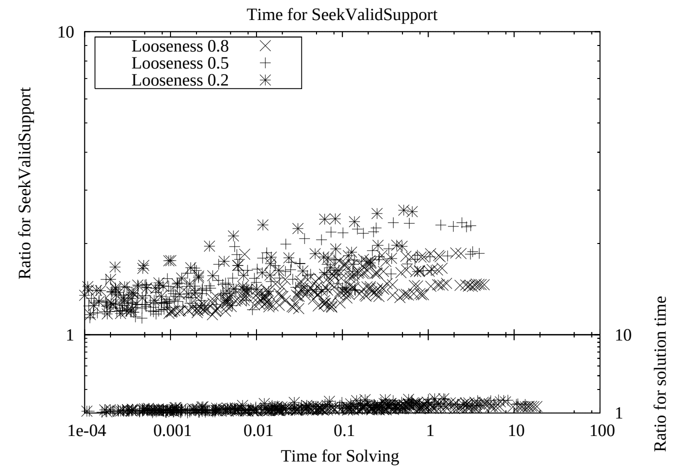

Figure 6: Comparison of using Tries and Hologram. We plot time using Hologram on the x-axis, and ratio of that technique to using Tries on the y-axis.

The results above show that Tries is the best performing of the techniques we have studied on this data set.[3] We have examined Tries' behaviour in more detail, but omit plots for reasons of space. At all densities, we found two interesting trends. 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 behaviour can be summarised by saying that as the size of the tables increases, the benefits of using Tries increases too.

GAC using Watched Literals

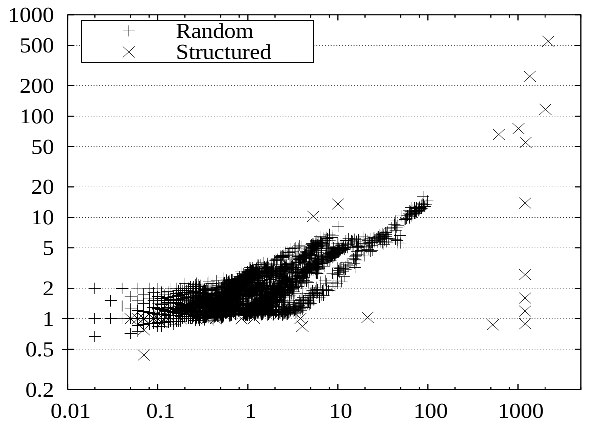

Figure 7: Comparison of performance in Minion of using Tries with the Simple data structure on all random and structured instances. The x-axis shows the run time using the comparison technique in secs, while the y axis shows the speedup factor in nodes in the search tree searched per second using Tries. We report nodes per sec because some non-random instances timed out after 1200secs.

The constraint solver Minion (Gent, Jefferson, & Miguel 2006b) allows the use of "watched literals" as another means of constraint propagation (Gent, Jefferson, & Miguel 2006a). Gent et al provide a variant of GAC-2001 (Bessière *et al.* 2005) adapted for watched literals, and report on its integration into Minion. Implementing GAC for table constraints with watched literals has some potential advantages. For example, implementation is comparatively simpler than GAC-schema, but is still a 'fine-grained' algorithm, i.e. propagation only happens when particular values lose support (Gent, Jefferson, & Miguel 2006a). The algorithm requires SeekValidSupport, just as GAC-schema does, so we can use the existing and new data structures.

[3]Unfortunately we have not implemented Lecoutre & Szymanek's technique for GAC-Schema.

We have implemented the data structures in Minion, integrating each in turn with table constraints using watched literals. Minion is open source so available for researchers to modify in this way. As well as the methods from the previous section, we implemented Lecoutre and Szymanek's method. We used a set of identically specified iMacs, with Intel Core Duo 2GHz processors and 2GB RAM under OS X 10.4.7. We built our variants using gcc 4.0.1 using flags -fomit-frame-pointer -O3 -march=pentium-m -mdynamic-no-pic. We used the same set of random instances described above (excluding some where input files reached 20MB). We also tested non-random, structured, instances which use table constraints from the problem classes graceful graphs, semigroup construction, BIBD's, the prime queens problem, and Golomb rulers. We experimented on 38 non-random instances. While this compares unfavourably with the thousands of random instances, structured instances are much harder to come by. Even 38 instances is many more than the structured instances reported in comparable studies (Lecoutre & Szymanek 2006; Lhomme & Regin 2005).

Figure 7 shows performance using Tries against the Simple method on our benchmark set. Speed is almost always improved, and very often by a factor of 10, especially on more difficult instances. The largest win is a speedup of more than 500-fold. Figure 8 shows results against each individual non-random class. No clear pattern emerges. Very dramatic improvements are obtained, while on some instances we see a marginal slowdown. The fact that we have not found instances where significant slowdowns occur is important.

We found that the best performance was using Tries. Lhomme and Régin's Hologram could be more than 20 times slower than Tries, although outperforming Tries by up to 2 times on some of the simpler instances taking up to 5 seconds. This is seen in Figure 9. Hologram was usually faster than Simple, although some instances could be 10 times slower using Hologram than Simple. Lecoutre and Szymanek's method is also effective and much faster than Simple, but is almost always slower than Tries. This is shown in Figure 10. However, it is never 5 times slower than Tries. For Next-Difference lists, performance was better with just one list. Next-Difference lists (with one list) often outperformed Tries, by up to 5 times, but not on instances requiring more than 10 seconds. On harder instances, Tries could in turn outperform Next-Difference lists by 5 times. We regard Tries as more successful, since speedups are more important as run times become longer. Lhomme and Régin's Hologram could be more than 20 times slower than Tries, although outperforming Tries by up to 2 times on some of the simpler instances taking up to 5 seconds. Hologram was usually faster than Simple, although some instances could be 10 times slower using Hologram than Simple.

The Simple method is no straw man: the implementation we compare with is in Minion, one of the fastest constraint solvers, using the implementation of GAC suggested by Minion's authors as being particularly attuned to its design (Gent, Jefferson, & Miguel 2006a). Note that all the comparisons, including 500-fold speedups, are achieved while searching the same tree, since all methods establish GAC

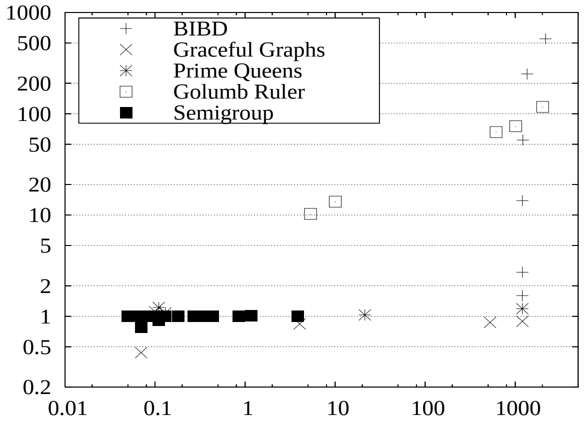

Figure 8: Comparison of performance in Minion of using Tries with the Simple data structure on structured instances divided into classes. Axes are as in Figure 7.

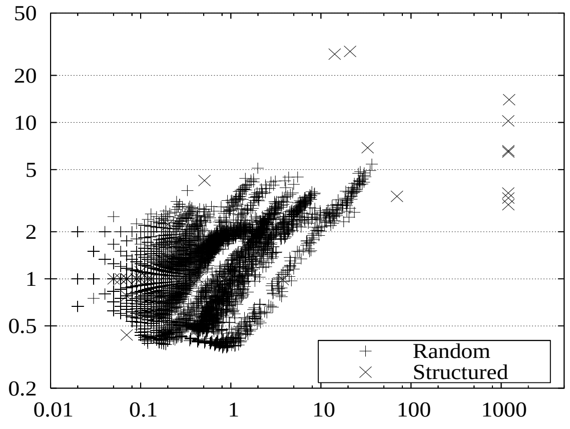

Figure 9: Comparison of performance in Minion of using Tries with the Hologram data structure. Axes are as in Figure 7.

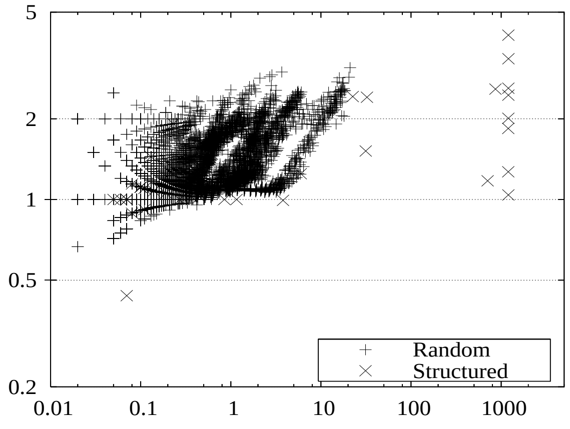

Figure 10: Comparison of performance in Minion of using Tries with Lecoutre & Szymanek's method. Axes are as in Figure 7.

at each node. Also, since we did not measure time taken in SeekValidSupport, times incorporate all aspects of the solver, and not just GAC for table constraints.

Related work

Lhomme and Régin (2005) use the *Hologram* data structure to store tuples as one list. For each tuple, for each value, there is a pointer to the next tuple to contain the same value. For each tuple, there is a *redundant second tuple* alongside it with the equivalent set of pointers forward. The values of the redundant tuple are incremented modulo d from the previous redundant tuple. Therefore in d steps one can find a pointer to the next tuple to contain any value in any position. While this enables significant efficiency gains, the complexity of the search algorithm may be a problem.

Lecoutre and Szymanek (2006) give an algorithm seekSupport-valid+allowed, which uses binary search rather than simple iteration. Lists for each literal are constructed as in the basic algorithm. All lists are sorted in lexicographic order (lex, $\prec_{lex}$). The algorithm first constructs the lex least valid tuple t, then performs a binary search for t in the list, finding t' which is the lex least tuple in the list s.t. $t \prec_{lex} t'$. If t' is valid, we are done. Otherwise, the lex least tuple t'' is constructed s.t. $t' \prec_{lex} t''$ and t'' is valid. The algorithm then repeats the binary search and proceeds from there. Despite the expense of the binary search, this technique performs very well in our experiments.

Carlsson (2006) uses a directed acyclic graph (DAG) to represent the constraint, and gives an algorithm for propagation. Compared to Tries, the DAG is potentially much smaller. Also, ranges of values are represented in the DAG, and support for a range of values can be found in one step. Each time the propagator is called, support for all values is found by searching the DAG once. If the user provides a small DAG, the Carlsson algorithm should be very efficient. However, we presumably do not get a small DAG from arbitrary set of tuples. If the DAG is large, two problems are apparent: a backtracking bit is used for each node of the DAG, so a large amount of backtracking memory is required; and our optimization of using a trie for each variable is not present so it may be inefficient to find support for rightmost variables. It would be interesting to explore the use of DAGs constructed from arbitrary sets of tuples.

Cheng and Yap (2006) use ROBDDs to store and manipulate a constraint as a set of tuples, reducing the ROBDD as variables become assigned during search. The major limitation of this work is that it works only on binary variables.

We are not aware that tries have been used for GAC before. Zhang and Stickel (2000) used tries for the closely related problem of storing clause sets efficiently for unit propagation. Indeed this partly inspired our use of tries here.

Conclusions

This paper has explored the use of data structures for tuple search in table constraints, with a large experimental analysis. The Tries data structure has been successfully applied to this problem. Tries work well with both the watched-literal GAC algorithm and GAC-Schema. The novel Next-Difference lists are a simple idea and effective, though less successful than Tries in the watched-literal GAC algorithm. Next-Difference with one list has a lower space complexity than Tries, so remains of significant interest.

We have performed by far the most extensive set of experiments to date on the emerging new data structures for tuple search. We have confirmed the good experimental results of Lecoutre and Szymanek (2006), and our experiments place their algorithm between Tries and Next-Difference lists in terms of performance, with Tries performing best of the three. We have also confirmed the benefit of Lhomme and Régin's algorithm compared to the simple approach. Some of our detailed conclusions may be subject to revision if optimisations for particular techniques are found. But we can certainly conclude that the use of these data structures can speed up propagating table constraints considerably.

Finally, table constraints are widely applicable, and very significant improvements can be made over the simple algorithm, so we believe this area of research to be fruitful. We have only scratched the surface of the space of data structures. Further novel data structures could make significant improvements over Tries.

References

Bessiere, C., and Regin, J.-C. 1997. Arc consistency for general constraint networks: Preliminary results. In *Proc. IJCAI-97*, 398–404.

Bessière, C.; Régin, J.-C.; Yap, R. H. C.; and Zhang, Y. 2005. An optimal coarse-grained arc consistency algorithm. *Artif. Intell.* 165(2):165–185.

Carlsson, M. 2006. Filtering for the case constraint. Talk given at Advanced School on Global Constraints, Samos, Greece.

Cheng, K. C. K., and Yap, R. H. C. 2006. Maintaining generalized arc consistency on ad-hoc n-ary boolean constraints. In *ECAI-06*, 78–82.

Fredkin, E. 1960. Trie memory. *Comms. ACM* 3(9):490–499.

Gent, I. P.; Jefferson, C.; and Miguel, I. 2006a. Watched literals for constraint propagation in Minion. In *Proc. CP-2006*, 182–197. Springer.

Gent, I.; Jefferson, C.; and Miguel, I. 2006b. Minion: A fast, scalable constraint solver. In *Proc. ECAI-06*, 98–102.

Lecoutre, C., and Szymanek, R. 2006. Generalized arc consistency for positive table constraints. In *Proc. CP-2006*, 284–298.

Lhomme, O., and Regin, J.-C. 2005. A fast arc consistency algorithm for n-ary constraints. In *Proc. AAAI-05*, 405–410.

Mackworth, A. 1977. On reading sketch maps. In *Proc. IJCAI-77*, 598–606.

Wallace, M. 1996. Practical applications of constraint programming. *Constraints* 1(1/2):139–168.

Zhang, H. and Stickel, M.E. 2000. Implementing the Davis-Putnam Method. *J. Autom. Reasoning*, 24:277–296.