Optimizing Simple Tabular Reduction with a Bitwise Representation*

Ruiwei Wang[1,2], Wei Xia[3], Roland H. C. Yap[3] and Zhanshan Li[1,2]
[1]School of Software, Jilin University, Changchun, China
[2]Key Laboratory of Symbol Computation and Knowledge Engineering, Education Ministry, China
[3]School of Computing, National University of Singapore, Republic of Singapore
wangrw13@mails.jlu.edu.cn, {xiawei, ryap}@comp.nus.edu.sg, lizs@jlu.edu.cn

Abstract

Maintaining Generalized Arc Consistency (GAC) during search is considered an efficient way to solve non-binary constraint satisfaction problems. Bit-based representations have been used effectively in Arc Consistency algorithms. We propose STRbit, a GAC algorithm, based on simple tabular reduction (STR) using an efficient bit vector support data structure. STRbit is extended to deal with compression of the underlying constraint with c-tuples. Experimental evaluation show our algorithms are faster than many algorithms (STR2, STR2-C, STR3, STR3-C and MDDc) across a variety of benchmarks except for problems with small tables where complex data structures do not payoff.

1 Introduction

Constraint propagation is used for solving constraint satisfaction problems. The most well studied and successfully used constraint propagation technique is generalized arc consistency (GAC). Table constraints define constraint relations extensionally, i.e. the allowed combination of values, namely the most general form of a finite domain constraint. State-of-the-art GAC algorithms for table constraint include simple tabular reduction (STR) [Ullmann, 2007] algorithms: STR2 [Lecoutre, 2011], STR3 [Lecoutre *et al.*, 2012]; and Multi-valued Decision Diagram (MDD) algorithms: MDDc [Cheng and Yap, 2010], MDD4 [Perez and Régin, 2014]. Usually, GAC is invoked at each node in the backtrack search tree, giving rise to maintaining GAC.

Constraint and variable domain representation is the basis of a GAC algorithm. Several algorithms based on bit variable domains have been proposed for binary constraints. AC3^{bit} [Lecoutre and Vion, 2008], considered an efficient arc consistency (AC) algorithm optimizes AC3 [Mackworth, 1977] algorithm by using a bit variable domain to represent the binary constraint. MaxRPC^{bit} [Guo *et al.*, 2011] uses a bit variable domain to reduce the cost of searching for a PC-witness. In [Ullmann, 2010], domain reduction algorithms based on bit variable domain are given, such as AC, forward checking and focus domain reduction. A recent GAC algorithm using bit vectors for table constraints is implemented in the OR-Tools solver (see [Demeulenaere, 2015]). It maintains the validity of tuples with bit vectors and seeks support for variable domain values with bit operations. Our focus is exploiting bit-vectors in GAC for non-binary table constraints while gaining the benefits of simple tabular reduction.

When the size of a table constraint is large, constraint representations which reduce space can also substantially speed up the GAC algorithm. Various representations have been proposed to compress table constraints. For example, a compressed table [Katsirelos and Walsh, 2007] represents a set of tuples as c-tuples, a short-support [Jefferson and Nightingale, 2013] represents a set of tuples by omitting some variables which are implied, and a sliced table [Gharbi *et al.*, 2014] represents a set of tuples as a sub-table associated with patterns. These representations have been used in the following GAC algorithms. STR2-C and STR3-C [Xia and Yap, 2013] extends STR2 and STR3 for c-tables; shortSTR2 [Jefferson and Nightingale, 2013] extends STR2 with short support; and STR-slice [Gharbi *et al.*, 2014] enforces GAC on sliced tables. For problem instances with high compression, these algorithms can be faster than STR2.

In this paper, we introduce a new table constraint representation, called bit table. A bit table encodes the supports in the form of a dual table [Lecoutre *et al.*, 2015] with bit vectors. We propose a new GAC algorithm to maintain GAC during search, STRbit, using the bit table representation. STRbit like STR3 focuses on maintaining GAC during search (unlike STR2 which is a standalone GAC algorithm). The advantage of the bit table is that it allows operations which deal with maintaining support in the GAC algorithm to be performed in parallel using $O(1)$ machine instructions. STRbit can also be faster than STR3 as the bit table can be (much) smaller than the dual table. We also extend the bit table to handle compressed tables on c-tuples with the STRbit-C algorithm. Experimental evaluation using well known benchmarks show STRbit to be faster than the state-of-the-art STR2, STR3 and MDDc algorithms on most problem instances. In a similar way, STRbit-C is also faster than the corresponding c-table algorithms, STR2-C and STR3-C.

---

*This work was supported by the MOE AcRF (R-252-000-505-112, R-252-000-592-112), the NSFC (61272208, 61373052), and the Jilin Province Science and Technology Development Plan (20140101200JC).

	X	Y	Z
1	a	a	b
2	a	b	b
3	a	b	c
4	a	c	c
5	b	a	c
6	b	c	c
7	c	a	a
8	c	b	a

(a) Standard table

X		Y		Z	
a	1,2,3,4	a	1,5,7	a	7,8
b	5,6	b	2,3,8	b	1,2
c	7,8	c	4,6	c	3,4,5,6

(b) Dual table

Figure 1: The standard and dual table representation for constraint $C(X, Y, Z)$.

	index of original tuples
θ_1	1, 2, 3, 4
θ_2	5, 6, 7, 8

(a) A table partition

X		Y		Z	
a	$(\theta_1, 1111)$	a	$(\theta_1, 1000), (\theta_2, 1010)$	a	$(\theta_2, 0011)$
b	$(\theta_2, 1100)$	b	$(\theta_1, 0110), (\theta_2, 0001)$	b	$(\theta_1, 1100)$
c	$(\theta_2, 0011)$	c	$(\theta_1, 0001), (\theta_2, 0100)$	c	$(\theta_1, 0011), (\theta_2, 1100)$

(b) Bit table

Figure 2: A table partition and its corresponding bit table.

2 Background

A constraint satisfaction problem (CSP) is a pair $(\aleph, \zeta)$ where $\aleph$ is a set of n variables and ζ is a set of e constraints. Each variable $X \in \aleph$ has a domain $D(X)$ defining possible values for X. The maximum domain size of all variables is denoted by d. A *literal* (X, a) represents a variable value pair. During search $dom(X)$ denotes the current domain of X. If $a \in dom(X)$, we say that (X, a) is valid, otherwise (X, a) is invalid. Each constraint $C \in \zeta$ involves a constraint scope $scp(C)$ and a relation $rel(C)$, where $scp(C)$ is a subset of $\aleph$ and $rel(C)$ is a set of tuples satisfying constraint C. A tuple $\tau \in rel(C)$ is composed of a set of literals of the variable in $scp(C)$. The *number of literals* in $rel(C)$ is denoted by L. A solution to a CSP is a set of valid literals for all variables such that all constraint are satisfied.

A tuple $\tau \in rel(C)$ is a support of literal (X, a) iff $(X, a) \in \tau$. A tuple $\tau \in rel(C)$ is valid iff (X, a) is valid for any literal $(X, a) \in \tau$. A literal (X, a) is generalized arc consistent (GAC) on a constraint C involving X iff there exists a valid support τ of (X, a) on C. A variable X is GAC on a constraint C involving X iff $dom(X) \neq \emptyset$ and (X, a) is GAC on C for each value $a \in dom(X)$. A constraint C is GAC iff X is GAC on C for each $X \in scp(C)$. A CSP is GAC iff each of its constraints is GAC.

A positive (negative) table constraint C is a constraint whose relation $rel(C)$ is represented in a table of allowed (disallowed) tuples. A table can be transformed into an equivalent dual table by mapping the index of tuples to their supported literals. Figure 1(a) gives a (standard) table with 24 literals and its corresponding dual table in Figure 1(b).

3 Bit representation in STRbit

We first introduce a bit representation for table constraints. The idea of our bit representation is to encode the supports of each literal in a constraint's dual table with bit vectors. We assume a natural word size w where the underlying processor provides $O(1)$ time bit vector operations. We partition a table into a set of *sub-tables* such that the number of tuples in a sub-table equals w (w.l.o.g., we assume the total number of tuples in the table is divisible by w). Then for each literal (X, a) and sub-table θ, a bit vector $mask$ records the supports of (X, a) in θ. The ith bit in $mask$ indicates whether the ith tuple in sub-table θ is a support (value 1) or not (value 0).

Function GACInit(C : *Constraint*)

```
remove invalid tuples from $rel(C)$ and build a bit table
  for constraint $C$;
for each sub-table $\theta$ of $rel(C)$ do
    $VAL(C, \theta) := -1$;   // all bits are value 1
for each $X \in scp(C)$ and $a \in dom(X)$ do
    $LAST(C, X, a) := BIT\_SUP(C, X, a).size - 1$;
for each $X \in scp(C)$ do
    $DEL(C, X) := \emptyset$ ;
```

Algorithm 1: STRbit (C : *Constraint*)

```
$deleteInvalidTuple(C)$;
return $searchSupport(C)$;
```

Definition 1. *A sub-table θ is a table support of (X, a) iff at least one tuple in θ is a support of (X, a). This is equivalent to that at least one bit of the corresponding bit vector $mask$ is value 1.*

For each literal in the dual table, we replace its index of support tuples by a set of value pairs whose first value θ gives the index of the table support and the second value is the corresponding bit vector $mask$. We call such value pairs as *bit supports* of literals, and the dual table with bit supports as a *bit table*. For convenience, we say "a tuple in θ" to mean that the tuple is included in the sub-table with index θ.

Example 1. *Figure 2(a) shows a partition of the table given in Figure 1(a) that the table is partitioned into 2 sub-tables θ_1 and θ_2 and each sub-table contains 4 original tuples. Then we use a bit-vector $mask$ to encode every sub-table. Figure 2(b) gives the corresponding bit table. (Y, a) has two bit supports. The first bit support $(\theta_1, 1000)$ indicates the 1st tuple in sub-table θ_1 is a support of (Y, a). The second bit support indicates the 1st and 3rd tuples in sub-table θ_2 are supports.*

The number of bit supports in bit table is $O(L)$, but always $\leq L$. Thus, the bit table can be seen as a way of compressing the dual table (see L/L_{bit} in Section 5).

3.1 Maintaining GAC on bit table during search

We give a GAC algorithm STRbit for the table constraint represented in a bit table. Our algorithm adapts the simple tabular reduction (STR) algorithm which shrinks tables dynamically during search. STRbit maintains GAC during search by updating the validity of tuples and seeking supports for each value in the variable's current domains. Validity of tuples is also represented by bit vectors *VAL*, where bit value 1 indicates the corresponding tuple is valid and value 0 indicates

```
Function deleteInvalidTuple(C : Constraint)
    for each $X \in scp(C)$ do
        for $a \in DEL(C, X)$ do
            for $i := 0$ to $LAST(C, X, a)$ do
1               $\theta := BIT\_SUP(C, X, a)[i].ts$;
                $u := BIT\_SUP(C, X, a)[i].mask$ & $VAL(C, \theta)$;
                if $u \neq 0$ then
                    $save((C, \theta), VAL(C, \theta), restoreV)$;
2                   $VAL(C, \theta) := (\neg u)$ & $VAL(C, \theta)$;
        $DEL(C, X) := \emptyset$;
```

```
Function searchSupport(C : Constraint)
    for each $X \in scp(C)$ and $a \in dom(C, X)$ do
        $now := LAST(C, X, a)$;
        $\theta := BIT\_SUP(C, X, a)[now].ts$;
1       while $BIT\_SUP(C, X, a)[now].mask$ & $VAL(C, \theta) = 0$
         do
            $now := now - 1$;
            if $now = -1$ then
                $remove(C, X, a)$;
                if $dom(X) = \emptyset$ then
                    return false;
                break;
            $\theta := BIT\_SUP(C, X, a)[now].ts$;
2       if $now \neq LAST(C, X, a)$ then
            $save((C, X, a), LAST(C, X, a), restoreL)$;
            $LAST(C, X, a) := now$;
    return true;
```

invalid. For example, we can use two 4-bit vectors to represent the validity of tuples in the two sub-tables in Figure 2(a). If we assume only tuple 1 in θ_1 and tuple 7 in θ_2 are valid and all the other tuples are invalid, then the values of the vectors are (1000) and (0010). Before giving the details of STRbit, we first introduce its data structures.

- $BIT\_SUP(C, X, a)$ is an array of bit supports of literal (X, a) in the bit table of constraint C. In fact, $BIT\_SUP(C, X, a)[i]$ is the ith bit-support of (X, a). $BIT\_SUP(C, X, a)[i].ts$ is the index of table support and $BIT\_SUP(C, X, a)[i].mask$ is the corresponding bit vector $mask$. $BIT\_SUP(C, X, a).size$ is the number of bit-supports of (X, a) and used for initialization.
- $VAL(C, \theta)$ is a bit vector recording the validity of tuples in θ.
- $LAST(C, X, a)$ records the last position i such that the table support $BIT\_SUP(C, X, a)[i].ts$ includes at least one valid tuple, and $BIT\_SUP(C, X, a)[j].ts$ doesn't include valid tuples for each $j > i$.
- $DEL(C, X)$ is a set of literals that have been deleted from $dom(X)$, but GAC has not been maintained on C.
- $restoreV$ and $restoreL$ are two stacks recording the validity of tuples and the last position of bit supports to restore information when backtracking happens.

```
Function remove(C, X, a)
    remove literal $(X, a)$ from $dom(X)$;
    add $X$ to the propagation queue;
    for each $C_1 \in \zeta$ and $C_1 \neq C$ and $X \in scp(C_1)$ do
        add $a$ to $DEL(C_1, X)$;
```

```
Function save(key, newdata, store)
    if $(key, olddata) \notin top(store)$ for any $olddata$ then
        add $newdata$ to $top(store)$
```

Like STR3, STRbit is only used to maintain GAC during search, as such, a separate GAC algorithm needs to be invoked before search to make the input instance GAC. Then we call the function *GACinit* to remove invalid tuples and initialize the data structures of STRbit before search starts. In *GACinit*, we set $VAL(C, \theta)$ to be a bit vector of ones (line 2) and set the separator $LAST(C, X, a)$ pointing to the last bit support (line 3). We then initialize $DEL(C, X)$ to the empty set since all constraints are GAC at the start.

During search, the STRbit algorithm is invoked for each constraint C when a value is removed from the domain of a variable involved in C. Algorithm 1 gives both parts of STRbit. Firstly, $deleteInvalidTuple(C)$ deletes all invalid tuples and updates bit vector *VAL*. For each removed literal (X, a) in *DEL*, $deleteInvalidTuple(C)$ set its bit supports to be invalid (between line 1 and 2). Then the second part $searchSupport(C)$ seeks supports for each (X, a) such that $a \in dom(X)$. We seek supports from the position recorded by *LAST(C,X,a)* and use now to point to the bit support being checked. θ indicates the index of sub-table so that we can fetch the validity of tuples in $VAL(C, \theta)$. We apply bit operations to check whether the current bit support includes a valid tuple (line 1). If all values are 0, there is no valid tuple in the current bit support and now shifted to the left by one. If $now = -1$, then there no valid bit tuple supports (X, a) and (X, a) can be deleted. The function $save$ is used to record the old data in a stack before updating the data structures. The data structure $restoreL$ records the old data of $LAST(C, X, a)$, and $restoreV$ records old data of $VAL(C, \theta)$. Upon backtracking, the algorithm just needs to reset the value of $LAST(C, X, a)$ and $VAL(C, \theta)$ by popping the old data from $restoreL$ and $restoreV$. In addition, since the constraint network is always consistent before accessing the next layer in the search tree, we just need to set $DEL(C, X)$ as empty when backtracking.

3.2 Complexity analysis

STRbit is designed to maintain GAC during search, thus, we analyse the "time complexity along a path" in the search tree. The time cost of STRbit is bounded by the number of bit supports included in the bit table. For one constraint C, L_{bit} denotes the number of bit supports included in the bit table.

Theorem 1. *The accumulated time cost of* r *arity constraint* C *in STRbit along a single path of length* m *in the search tree is* $O(L_{bit} + r^2 d^2 + m)$.

	X	Y	Z
1	a	a, b	b
2	a	b, c	c
3	b	a, c	c
4	c	a, b	a

(a) Standard c-table

	Index of original c-tuple
η_1	1, 2, 3, 4

(b) A c-table partition

X		Y		Z	
a	$(\eta_1, 1100)$	a	$(\eta_1, 1011)$	a	$(\eta_1, 0001)$
b	$(\eta_1, 0010)$	b	$(\eta_1, 1101)$	b	$(\eta_1, 1000)$
c	$(\eta_1, 0001)$	c	$(\eta_1, 0110)$	c	$(\eta_1, 0110)$

(c) Bit c-table

Figure 3: A c-table partition and its corresponding bit c-table.

Proof. The primary time cost of STRbit includes two parts. One is the function $deleteInvalidTuple(C)$: the time cost of this part is $O(L_{bit})$, since every bit support is only processed once; Another part is the function $searchSupport(C)$: the time cost of this part is $O(L_{bit}+r^2d^2)$. At line 2, when *LAST* is decreased, each bit support in *BIT_SUP* will be checked only once, so the time cost is $O(L_{bit})$. Otherwise when *LAST* is not changed, an invocation's cost is $O(rd)$, and the number of calls STRbit on C is at most rd times along a single path in search tree, so the time cost is $O(r^2d^2)$. Correspondingly, the total time cost is $O(L_{bit}+r^2d^2+m)$ along a single path in search tree, as all other statements have fixed cost $O(1)$ at each node. □

STR3 has a path-optimality property but STRbit is not (Theorem 1) so in the worst case a bit support in a bit table may be processed multiple times in a search tree path. From a practical efficiency perspective, the importance of path-optimality is unclear, e.g. STR2 is not path-optimal but is faster than STR3 on many benchmarks [Lecoutre *et al.*, 2012]. In practice, the actual cost in STRbit depends on the size of L_{bit} which can be much smaller than L (used in STR3) by a factor of w (due to bit vectors). Our experiments (Figure 4(a)) show STRbit to be up to 25X faster than STR3. The speedup ratio is not always close to the ratio L/L_{bit} because in STR algorithms L varies dynamically due to the tabular reduction, thus, the worst case time complexity may be a rather crude bound.

4 Bit-C representation in STRbit-C

We now combine Cartesian product compression with its bit representation giving a new dual table representation, called *bit c-table*. For an r-arity constraint $C(x_1, \ldots, x_r)$, the Cartesian product representation of a set of tuples $(\{a_{1,1}, \ldots, a_{1,k1}\}, \ldots, \{a_{r,1}, \ldots, a_{r,kr}\})$ is called a *c-tuple* [Katsirelos and Walsh, 2007]. This c-tuple admits any set of assignments that assigns one of $a_{1,1}, \ldots, a_{1,k1}$ to x_1, one of $a_{2,1}, \ldots, a_{2,k2}$ to x_2, etc. A c-tuple τ_c is a *c-support* of literal (X, a) iff there is a valid tuple in τ_c supports (X, a). We can compress a standard table into a *c-table* with c-tuples. The standard table in Figure 1(a) represented in c-table form is shown in Figure 3(a). As before, we partition a c-table into a set of *sub-c-tables* by the word size, where a sub-c-table is a subset of c-table.

Similar to table support, a sub-c-table η is a *table c-support* of (X, a) iff at least one c-tuple in η supports (X, a). Then for each literal (X, a), we use a set of pair $(\eta, mask)$ to represent a *bit c-support* of (X, a), where η is an index of table c-support and $mask$ is a bit vector. In $mask$, the ith bit indicates whether the corresponding ith c-tuple in η is a c-support of (X, a) (value 1) or not (value 0). For convenience, we say "a c-tuple in η" to mean that the c-tuple is included in the sub-c-table with index η.

```
Function deleteInvalidCTuple($C$ : Constraint)
      $presize := CDENSE[C].size$;
      for each $X \in scp(C)$ do
1         if $DEL(C,X) \neq \emptyset$ then
              $DEL(C,X) := \emptyset$;
              clear $BUFF$;                    // set all to 0
              for each $a \in dom(X)$ do
                  for $i := 0$ to $LAST(C,X,a)$ do
                      $\eta := BIT\_SUP(C,X,a)[i].ts$;
2                     $u := BIT\_SUP(C,X,a)[i].mask \mid BUFF[\eta]$;
3                     $BUFF[\eta] := u$;
              for each $\eta \in CDENSE[C]$ do
4                 $intersection := VAL(C,\eta) \& BUFF[\eta]$;
                  if $intersection \neq VAL(C,\eta)$ then
                      $save((C,\eta), VAL(C,\eta), restoreV)$;
5                     $VAL(C,\eta) := intersection$;
                  if $intersection = 0$ then
                      // deleting $\eta$
6                     $CDENSE[C] := CDENSE[C]/\{\eta\}$;
      if $presize \neq CDENSE[C].size$ then
          $save(C, CDENSE[C], restoreC)$;
```

Example 2. *Figure 3(b) gives a partition of the c-table $rel_c(C)$ in Figure 3(a). The partition has only one sub-c-table η_1 with all 4 tuples in $rel_c(C)$. Thus each domain value has at most one bit c-support, e.g. (Y, c) has one bit c-support $(\eta_1, 0110)$ in Figure 3(c). The value $(\eta_1, 0110)$ means that the second and third c-tuple in η_1 are (Y, c)'s c-supports.*

4.1 Maintaining GAC on bit c-table during search

We now give the STRbit-C GAC algorithm which works on bit c-tables. As with STRbit (Algorithm 1), STRbit-C is also composed of two functions: $deleteInvalidCTuple(C)$ to delete invalid c-tuples and $searchSupport(C)$ to seek supports for literals. A c-tuple is valid iff at least one ordinary tuple of the c-tuple is valid; otherwise it is invalid. Function $searchSupport(C)$ is the same as STRbit but $deleteInvalidCTuple(C)$ is different due to differences in how invalid c-tuples are deleted.

The data structures of STRbit-C is similar to STRbit. $VAL(C, \eta)$ represents the validity of c-tuples in η, and a sub-c-table η is valid iff $VAL(C, \eta) \neq 0$. $BIT\_SUP(C, X, a)$ represents a set of bit c-supports of literal (X, a) (array). Then we use three additional data structures:

- $CDENSE[C]$ is a set of index of valid sub-c-tables for constraint C, $CDENSE.size$ is the size of $CDENSE[C]$.
- $restoreC$ is a stack recording old data of $CDENSE[C]$ to restore information when backtrack happens.

	Instances	#	STRbit	STRbit-C	STR2	STR2-C	STR3	STR3-C	MDDc	L/L_{bit}	L_c/L_{bitc}	L/L_c	L/L_{bitc}	$avgP$	L	L_c
R	rand-3	50	16.74	**12.15**	52.56	31.54	41.12	38.95	29.18	6.25	12.30	2.35	28.92	0.0567	8K	3K
	rand-3-fcd	50	8.50	**6.09**	29.05	15.77	20.93	19.63	14.77	6.25	12.30	2.35	28.92	0.0573	8K	3K
	rand-8	20	8.43	9.18	**8.20**	10.40	93.85	94.52	13.78	22.11	22.76	1.54	35.00	0.0018	624K	406K
	rand-5-8X	26	70.88	19.71	416.36	112.70	467.80	226.85	**19.43**	13.01	23.62	3.35	79.15	0.0075	497K	148K
	rand-5-4X	50	6.85	**5.10**	61.22	33.92	29.38	26.69	8.68	11.28	15.93	2.54	40.41	0.0406	248K	98K
	rand-5-2X	50	**1.97**	2.64	10.02	11.35	5.22	8.02	4.23	10.62	11.80	1.79	21.09	0.0911	124K	69K
	rand-5	50	**2.02**	3.58	21.90	30.43	4.77	10.57	15.36	8.77	9.18	1.34	12.29	0.2379	62K	46K
	rand-10-60	31	**12.71**	22.18	141.81	268.66	40.51	130.40	29 time-out	4.32	4.34	1.00	4.34	0.2637	512K	511K
	dag-rand	25	12.39	18.38	**9.86**	21.82	198.44	217.87	72.69	29.65	29.99	1.04	31.07	0.0014	2M	2M
	half	16	70.46	**36.99**	233.32	98.07	9 time-out	242.75	57.69	22.15	25.90	5.65	146.28	0.0059	277K	49K
	MDD0.7	6	32.08	**11.93**	392.67	44.96	382.38	63.29	20.56	22.11	26.98	12.42	335.06	0.0197	273K	22K
	MDD0.9	9	3.78	**2.22**	53.40	3.61	30.82	3.92	2.80	22.11	28.25	34.53	975.32	0.0675	273K	7K
	bdd-small	35	2.30	**2.23**	3.21	4.68	17.18	18.00	11.27	40.99	41.86	1.25	52.49	0.0385	1M	829K
	bdd-large	35	3.09	**3.07**	8.70	9.41	42.68	41.43	30.75	38.13	39.01	1.24	48.41	0.0421	103K	83K
S	golombR	16	42.04	**30.12**	50.04	152.53	70.93	47.60	31.46	6.62	30.20	5.54	167.46	0.0206	566K	103K
	uk	29	**2.16**	2.65	7.29	13.45	6.64	8.16	45.83	5.59	5.42	1.16	6.34	0.1806	105K	92K
	ogd	29	**0.98**	1.39	9.70	33.96	4.89	8.46	27.26	7.52	7.09	1.25	8.92	0.2249	199K	165K
	lex	40	0.84	**0.82**	1.76	2.68	1.68	1.84	6.39	4.48	4.44	1.06	4.77	0.1064	12K	12K
	words	40	**2.40**	2.73	6.35	10.03	5.74	6.08	24.76	5.27	5.18	1.11	5.77	0.1181	26K	24K
	lemma	9	10.79	**10.61**	15.46	11.53	30.91	20.89	11.80	9.73	13.51	2.36	31.92	0.077	957	405
	modR	50	7.28	**6.74**	17.58	18.12	17.24	18.04	18.01	12.57	10.06	22.60	227.56	0.0396	12K	542
	cril	1	3.48	**3.07**	4.69	3.21	6.07	3.16	3.65	29.69	14.88	43.70	650.31	0.0563	12K	284
	allInter	11	**15.02**	15.63	16.28	16.52	16.58	16.69	15.98	5.30	5.35	1.18	6.32	0.0603	379	321
	tsp-20	15	1.49	1.69	**1.20**	1.51	1.77	2.14	1.61	1.55	1.55	1.00	1.55	0.002	41K	41K
	tsp-25	15	14.90	16.38	**12.48**	14.71	17.73	19.75	16.28	1.55	1.56	1.00	1.56	0.0021	41K	41K
M	aim-100	17	49.08	50.68	48.78	54.49	48.85	48.13	**43.68**	3.47	2.24	1.55	3.47	0.4967	20	13
	aim-200	8	52.76	57.81	49.74	50.28	53.51	50.87	**42.21**	3.48	2.24	1.55	3.48	0.7299	20	13
	aim-50	24	0.18	0.17	0.18	**0.16**	**0.16**	0.17	**0.16**	3.47	2.23	1.55	3.47	0.4549	20	13
	dubois	5	183.67	181.25	175.90	203.01	173.17	176.09	**167.12**	2.00	2.00	1.00	2.00	0.283	12	12
MC	cc	8	53.85	56.56	61.99	61.73	59.26	53.26	**46.08**	9.14	4.01	2.58	10.35	0.4441	88	34
	ramsey	5	6.24	6.30	7.68	7.77	6.65	**5.66**	5.91	11.29	3.80	2.97	11.29	0.3745	115	38
	jnh	50	0.49	0.51	0.63	**0.47**	0.62	0.51	0.51	30.93	6.04	12.58	76.05	0.2736	787	62
	ii	10	12.51	6.98	1 time-out	6.69	1 time-out	6.73	**6.50**	42.47	10.84	458.13	5K	0.2704	78K	170
	PH-k-j	9	2.69	2.18	39.58	2.31	13.01	**2.01**	2.52	19.81	1.00	84K	84K	0.2363	4M	48

Table 1: Mean results of GAC algorithms. “cc”, “modR”, “lemma”, “golomb”, “uk, ogd, lex, words” and “allInter” stand for “chessboardColoration”, “renault and renault-mod”, “Schur’s lemma”, “golomb ruler”, “crosswords-uk, ogd, lex, words” and “all interval series”.

- *BUFF* is a set of bit vectors for computing the validity of c-tuples.

If a c-tuple τ_c in sub-c-table η is valid, then for each variable X in $scp(C)$, there must be a valid literal (X, a) and a bit c-support $(\eta, mask) \in \mathit{BIT\_SUP}(C, X, a)$ such that the corresponding bit in $\mathit{VAL}(C, \eta)$ & $mask$ for τ_c $((\mathit{VAL}(C, \eta)\ \&\ mask)[\tau_c])$ is value 1. This means that the set of valid c-tuples for constraint C equals

$$\bigcap_{X\in scp(C)} \bigcup_{a\in dom(X)} \bigcup_{(\eta,mask)\in \mathit{BIT\_SUP}(C,X,a)} \psi(\eta, mask)$$

where $\psi(\eta, mask)=\{\tau_c \in \eta \mid (\mathit{VAL}(C, \eta)\ \&\ mask)[\tau_c] = 1\}$. At line 1, if $\mathit{DEL}(C, X) = \emptyset$, no literal has been deleted after the last invocation of $deleteInvalidCTable(C)$, so variable X can be skipped. Otherwise, for each value $a \in dom(X)$ and bit c-support $\mathit{BIT\_SUP}(C, X, a)[i]$ such that $i \in [0, \mathit{LAST}(C, X, a)]$, $\mathit{BIT\_SUP}(C, X, a)[i].mask$ is recorded in *BUFF* at line 2 and line 3 (bitwise OR). Then for each $\eta \in \mathit{CDENSE}[C]$, $\mathit{VAL}(C, \eta)$ is updated by the bitwise AND operation of $\mathit{VAL}(C, \eta)$ and $\mathit{BUFF}[\eta]$ between lines 4 and 5. In addition, η will be deleted from $\mathit{CDENSE}[C]$ if $\mathit{VAL}(C, \eta)$ equals 0 (line 6).

4.2 Complexity analysis

Similar to STRbit, the time cost of STRbit-C along a path in search tree depends on the number of bit c-supports in bit c-table. We use L_{bitc} to represent the number of bit c-supports in a c-table.

Theorem 2. *The accumulated time cost of r-arity constraint C in STRbit-C along a single path of length m in the search tree is $O(dL_{bitc} + r^2d^2 + m)$.*

The proof is similar to Theorem 1 and omitted here.

5 Experiments

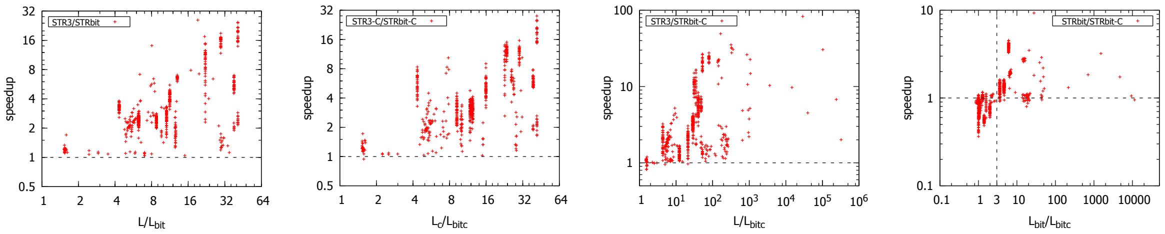
We evaluate STR2, STR3, STR2-C, STR3-C, MDDc, STRbit and STRbit-C in the Abscon [Merchez *et al.*, 2001] solver. Our implementation uses 64-bit numbers (`long` Java type) so $w = 64$ and partitions the table in $lexico$ order. In addition, we extract the c-table from the MDD. Experiments are run on a 3.40 GHz Intel core i7 processing on Linux, and all algorithms use the $dom/ddeg$ variable ordering heuristic and $lexico$ value ordering heuristic. Timeout is 600 seconds. We consider classical series instances[1], the series[2] introduced in STR2-C, and the PH-k-j series used in STR3 (896 instances).

Table 1 gives the mean runtime (in seconds) of 7 algorithms on different benchmarks. The column # gives the number of instances in each series. L (L_c) is the mean number of literals in the standard tables (c-tables), and L_{bit} (L_{bitc}) is the mean number of bit supports (bit c-supports) in the bit tables (bit c-tables). The columns L/L_{bit}, L_c/L_{bitc}, L/L_c and L/L_{bitc} represent the corresponding compression ratio. The column $avgP$ (from the STR3 evaluation [Lecoutre *et al.*, 2012]) is the mean ratio of “the number of tuples in the current standard table during search” to “the number of tuples in the initial standard table” during search. In addition, we divide the benchmarks into 4 groups: R for random benchmarks; S for structured; M for the instances with a small average table size ($L < 50$); and MC for small c-table sizes ($L_c < 200$).

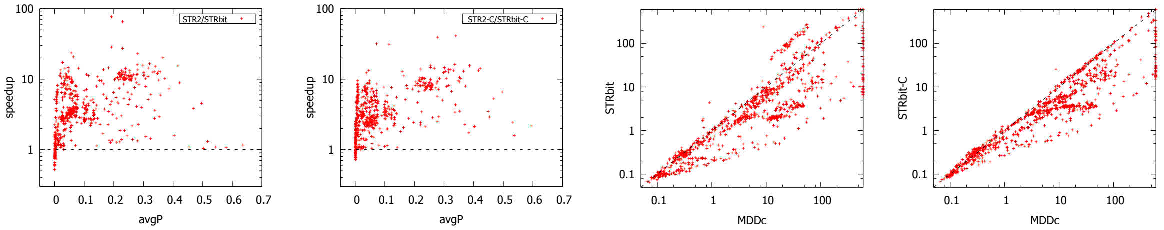
---

[1] http://www.cril.univ-artois.fr/%7Elecoutre/benchmarks.html

[2] http://www.comp.nus.edu.sg/%7Exiawei/STRC-benchmarks/



(a) STR3 vs STRbit (b) STR3-C vs STRbit-C (c) STR3 vs STRbit-C (d) STRbit vs STRbit-C



(e) STR2 vs STRbit (f) STR2-C vs STRbit-C (g) STRbit vs MDDc (h) STRbit-C vs MDDc

Figure 4: Comparison between different algorithms.

For benchmarks in groups R, S and *MC*, STRbit is faster than STR3 and the compression ratio L/L_{bit} is large. For example, in the MDD0.7 and MDD0.9 series, $L/L_{bit} > 20$ so STRbit can be 10X faster than STR3. STRbit is also faster than STR2 on most series except the ones with low $avgP$. We highlight the effect of the $avgP$ factor which measures the average reduced table size (by tabular reduction) during search. For example, on the tsp-20, tsp-25, rand-8 and dag-rand series, $avgP < 0.004$ so STRbit is a little slower than STR2 but still faster than STR3 (for dag-rand it is 16X faster). While for the rand-10-60 series, $avgP > 0.2$, so STRbit can be faster than STR2 by an order of magnitude. Similarly, on most series in the R and S group, STRbit-C is faster than STR2-C and STR3-C and MDDc. The instances in the M group have small average table sizes, so fixed costs (data structure initialization, restoration, etc.) during search can dominate. The MDDc algorithm, which maintains only one sparse set during search, is fast on these instances, while STRbit is close to STR2 and STR3. In addition, on the series in the *MC* group, the compression of c-table makes the size small (< 200), hence the time cost of STRbit-C is close to STR2-C, STR3-C and MDDc on these series.

In Figure 4, we present scatter plots comparing the speedup of STRbit and STRbit-C with various factors. Every dot in these graphs corresponds to an instance. For Figures 4(a), 4(b), 4(c), 4(d), 4(e) and 4(f), we avoid some "noise" by removing the instances solved within 0.5 second (the slowest algorithm) and the series in M. We also omit the series in MC for Figures 4(b) and 4(f), because the size of the c-table is very small. Figures 4(a), 4(b), 4(c) and 4(d) depict the impact of compression ratio on the performance of STRbit and STRbit-C. As we can see, when the compression ratios L/L_{bit}, L_c/L_{bitc} and L/L_{bitc} increase, STRbit (STRbit-C) can be up to 25X (80X) faster than STR3, while STRbit-C can be up to 27X faster than STR3-C. In addition, STRbit-C is faster than STRbit on most instances when $L_{bit}/L_{bitc} \geq 3$. On many instances, the STRbit (STRbit-C) compression ratios are larger than the speed up, we believe it is due to the initialization times and other costs during search. Furthermore, the ratio is only an estimate as the true compression ratio varies dynamically due to tabular reduction. Figures 4(e) and 4(f) show the effect of $avgP$. When $avgP$ is small (< 0.004), STRbit (STRbit-C) may be a little slower than STR2 (STR2-C), but when $avgP$ is larger, STRbit (STRbit-C) can be up to 70X (40X) faster than STR2 (STR2-C). Figure 4(g) and 4(h) compare STRbit and STRbit-C with MDDc using all series in Table 1. These two graphs show that our algorithms are faster than MDDc on most instances.

6 Conclusion

In this paper, we introduce new table representations, bit table and bit c-table. Then we propose two algorithms to maintain GAC during search on bit table and bit c-table. Experiments show that our algorithms outperform the state-of-the-art GAC algorithms STR2, STR3 and MDDc and for compressed tables also STR2-C and STR3-C. Previous work has shown a dichotomy for STR algorithms: (i) in tables where the tabular reduction is strong, small avgP, STR2 is very effective; (ii) but when the tables reduce less, larger avgP, then STR3 performs better. Our STRbit algorithms significantly narrow this gap and the benchmarks show that STRbit is more efficient than STR2 and STR3 in most cases even when the reduction rate is small. This suggests that STRbit algorithms can replace the combination of STR2 and STR3.

Recently, the algorithmic framework of STR has been extended in other directions. For example, the factor encoding [Likitvivatanavong *et al.*, 2014; 2015] adapts better to STR-based algorithms than MDD-based algorithms. The GIC4 algorithm [Bessiere *et al.*, 2013] enforcing global consistency for configuration problem also uses STR-based algorithms. There is potential for our algorithms along these directions.

References

[Bessiere *et al.*, 2013] Christian Bessiere, Héléne Fargier, and Christophe Lecoutre. Global inverse consistency for interactive constraint satisfaction. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming*, pages 159–174, 2013.

[Cheng and Yap, 2010] Kenil Cheng and Roland H. C. Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.

[Demeulenaere, 2015] Jordan Demeulenaere. Efficient algorithms for table constraints. Master's thesis, UCLouvain, 2015.

[Gharbi *et al.*, 2014] Nebras Gharbi, Fred Hemery, Christophe Lecoutre, and Olivier Roussel. Sliced table constraints: Combining compression and tabular reduction. In *Proceedings of the 11th International Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming*, pages 120–135, 2014.

[Guo *et al.*, 2011] Jinsong Guo, Zhanshan Li, Yonggang Zhang, and Xuena Geng. MaxRPC algorithms based on bitwise operations. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming*, pages 373–384, 2011.

[Jefferson and Nightingale, 2013] Christopher Jefferson and Peter Nightingale. Extending simple tabular reduction with short supports. In *Proceedings of the 23rd International Joint Conferences on Artificial Intelligence*, pages 573–579, 2013.

[Katsirelos and Walsh, 2007] George Katsirelos and Toby Walsh. A compression algorithm for large arity extensional constraints. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, pages 379–393, 2007.

[Lecoutre and Vion, 2008] Christophe Lecoutre and Julien Vion. Enforcing arc consistency using bitwise operations. *Constraint Programming Letters*, 2:21–35, 2008.

[Lecoutre *et al.*, 2012] Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland H. C. Yap. A path-optimal GAC algorithm for table constraints. In *Proceedings of the 20th European Conference on Artificial Intelligence*, pages 510–515, 2012.

[Lecoutre *et al.*, 2015] Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland H. C. Yap. STR3: A path-optimal filtering algorithm for table constraints. *Artificial Intelligence*, 220:1–27, 2015.

[Lecoutre, 2011] Christophe Lecoutre. STR2: optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.

[Likitvivatanavong *et al.*, 2014] Chavalit Likitvivatanavong, Wei Xia, and Roland H. C. Yap. Higher-order consistencies through GAC on factor variables. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming*, pages 497–513, 2014.

[Likitvivatanavong *et al.*, 2015] Chavalit Likitvivatanavong, Wei Xia, and Roland H. C. Yap. Decomposition of the factor encoding for CSPs. In *Proceedings of 24th International Joint Conference on Artificial Intelligence*, pages 353–359, 2015.

[Mackworth, 1977] Alan K. Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118, 1977.

[Merchez *et al.*, 2001] Sylvain Merchez, Christophe Lecoutre, and Frédéric Boussemart. Abscon: A prototype to solve csps with abstraction. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 730–744, 2001.

[Perez and Régin, 2014] Guillaume Perez and Jean-Charles Régin. Improving GAC-4 for table and MDD constraints. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming*, pages 606–621, 2014.

[Ullmann, 2007] Julian R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Sciences*, 177:3639–3678, 2007.

[Ullmann, 2010] Julian R. Ullmann. Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *Journal of Experimental Algorithmics*, 15:1–6, 2010.

[Xia and Yap, 2013] Wei Xia and Roland H. C. Yap. Optimizing STR algorithms with tuple compression. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming*, pages 724–732, 2013.