Neural Comput & Applic (2019) 31 (Suppl 1):S499–S508
DOI 10.1007/s00521-017-3015-7

ORIGINAL ARTICLE

Saving constraint checks in maintaining coarse-grained generalized arc consistency

Hongbo Li[1] · Ruizhi Li[1] · Minghao Yin[1]

Received: 8 December 2016 / Accepted: 11 April 2017 / Published online: 3 May 2017

Abstract Constraint check plays a central role in establishing generalized arc consistency which is widely used to solve constraint satisfaction problems. In this paper, we propose a new generalized arc consistency algorithm, called GTR, which ensures that the tuples that have been checked to be allowed by a constraint will never be checked again. For each constraint, GTR maintains a dynamic list of the tuples that were checked to be allowed by this constraint and check their validities to identify some values with supports. It is equipped with a mechanism avoiding redundant validity checks. The basic GAC3 algorithm is employed to find a support for the rest values and to add new tuples to the dynamic list. The experiments show that maintaining GTR during search saves a number of constraint checks. It also brings some improvements over cpu time while solving some CSPs with tight constraints.

Keywords Constraint satisfaction · Local consistency · Backtracking

1 Introduction

Constraint check plays a central role in arc consistency (AC) [16]. Maintaining arc consistency (MAC) [15, 18] is widely used to solve binary constraint satisfaction problems (CSP). In non-binary CSPs, AC is replaced by generalized arc consistency (GAC) [4]. An efficient MGAC algorithm usually has two features: (1) the GAC algorithm it uses is efficient, (2) it maintains few data structure during search. GAC algorithms can be classified into the coarse-grained and the fine-grained. The coarse-grained algorithms, such as AC3 and its improvements, are based on constraint-oriented propagation schemes. The latter [1, 3, 17] are based on value-oriented propagation schemes. The fine-grained algorithms usually maintain elaborate data structures during search, so the coarse-grained GAC algorithms are more efficient and more popular when being used in search. The original AC3 (GAC3) algorithm has the worst-case time complexity $O(ed^3)$ (O (er^3d^{r+1})). By recording *last* supports, AC3.1 algorithm avoids some repeated constraint checks and it has an optimal worst-case time complexity $O(ed^2)$ [5], but MAC3.1 is inefficient due to its heavy data structure. MAC3.2 [9] explores multi-directional supports, but it still maintains heavy data structures during search. MAC3r algorithm [14] explores residue supports which are not maintained during search, so it costs less time than MAC3 and MAC3.1. Making use of multi-directional residues, AC3rm algorithm [10] improves AC3r. MAC3rm is efficient to solve CSPs, although its worst-case complexity is $O(ed^3)$. This is because MAC3rm also maintains few data structure during search. Exploring multiple residues, MAC3rm2 [11] is more efficient than MAC3rm. All the coarse-grained AC algorithms can be extended to GAC versions.

In some problems, we may face tight constraints, such as the constraint C_{10} in *mknap-1-3* has a tightness higher than 0.99 and the weightedSum constraints in *magicSquare*[1] problems have tightness higher than 0.95. The GAC algorithms usually execute much more constraint checks to

✉ Minghao Yin
ymh@nenu.edu.cn

[1] School of Computer Science and Information Technology, Northeast Normal University, Changchun, 130117, China

[1]The two instances are downloaded from http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html.

enforce GAC on these tight constraints than dealing with loose constraints. For example, the *mknap-1-3* instance contains eleven 20-arity constraints and only one of them, C_{10}, has tightness higher than 0.99, the tightnesses of the other constraints are between 0.25 and 0.47. We use MGAC3rm2 to solve *mknap-1-3* and observe that the number of constraint checks executed on C_{10} is 1,067,710, whereas the total number on all the other constraints is 40,744. The result indicates that we should pay more attention to GAC algorithms for tight constraints, especially for those processed by a generic GAC algorithm. Although saving constraint checks does not always save time [20], it is important to save constraint checks in MGAC when constraint checks are relatively expensive. Some constraint checks are repeated in coarse-grained MGAC and [14] we identify two types of repeated constraint checks:

- Positive repeat: a constraint check for a tuple τ on a constraint c is performed and it is allowed by c. Later in the search, τ is still valid and it is checked again. Such repeated constraint checks are called positive repeats.
- Negative repeat: a constraint check for a tuple τ on a constraint c is performed and it is disallowed by c. Later in the search, τ is still valid and it is checked again. Such repeated constraint checks are called negative repeats.

In this paper, we propose a new generic GAC algorithm, called growing tabular reduction (GTR), which avoids all positive repeats and some of the negative repeats. It ensures that a tuple on a constraint c will never be checked again if it has been checked to be allowed by c. For each constraint c, it maintains a dynamic list recording the tuples that have been checked to be allowed by c. It first iterates over all active tuples in the list and checks their validities. A tuple is valid iff all the values in the tuple are present in the domains of the corresponding variables. If a tuple in the list is valid, all the values in this tuple have supports on c. Secondly, it uses constraint checks to seek supports for those values that have not found a support. If a new tuple is found to be allowed by c, it will be added into the list of c. There is extensive redundant work if we check all tuples in the list, so we propose a method to avoid redundant validity checks in maintaining GTR during search. An improved version, GTR2, is also introduced. The experimental results show that, compared with the classical coarse-grained MGAC algorithms, MGTR algorithms are not efficient on binary instances, but they save both constraint checks and cpu time on the non-binary instances with larger arity constraint that are tight.

This paper is organized as follows. Section 2 provides some technical background about CSP. The GTR algorithm and its improvement are introduced in Section 3. Section 4 presents the discussions and some related works. The experimental results and the analysis are in Section 5. Finally, conclusion and future work are in Section 6.

2 Background

A constraint satisfaction problem (CSP) P is a triple $P = \langle X, D, C\rangle$ where X is a set of n variables $X = \{x_1, x_2 ... x_n\}$, D is a set of domains $D = \{dom(x_1), dom(x_2) ... dom(x_n)\}$ where $dom(x_i)$ is a finite set of possible values for variable x_i, C is a set of e constraints $C = \{c_1, c_2 \ ... \ c_e\}$. A constraint c consists of two parts, an ordered set of variables $scp(c) = \{x_{i1}, x_{i2} \ ... \ x_{ir}\}$ and a subset of the Cartesian product $dom(x_{i1}) \times dom(x_{i2}) \times ... \times dom(x_{ir})$ that specifies the allowed (or disallowed) combinations of values for the variables $\{x_{i1}, x_{i2} \ ... \ x_{ir}\}$. $|scp(c)|$ is the arity of c. We use r to denote the arity of a constraint and d to denote the domain size of a variable. An element of $dom(x_{i1}) \times dom(x_{i2}) \times ... \times dom(x_{ir})$ is called a tuple on $scp(c)$, denoted by τ. $\tau[x]$ is the value of x in τ. The tightness of a constraint c is t/d^r, where t is the number of disallowed tuples on $scp(c)$ and d^r is the number of all tuples on $scp(c)$. Verifying if a given tuple is allowed by a constraint is called a constraint check and verifying if a given tuple is valid is called a validity check. (x, a) denotes the value a for variable x.

Definition 1 (Generalized arc consistency [4]) Given a CSP $P = \langle X, D, C\rangle$, a constraint $c \in C$, and a variable $x \in scp(c)$,

- A value (x, a) is consistent with c iff there exists a valid tuple τ allowed by c and $\tau[x] = a$. τ is called a support for (x, a) on c.
- A constraint c is generalized arc consistent iff $\forall x \in scp(c)$, $dom(x) \neq \emptyset$ and $\forall a \in dom(x)$, (x, a) is consistent with c.
- P is generalized arc consistent iff all the constraints of C are generalized arc consistent.

To establish GAC on a constraint, GAC algorithms seek a support for every value (x, a) on the constraints involving x and remove those values without any support on these constraints. If the domain of a variable is empty, GAC fails. To seek a support for a value on a constraint, the GAC-valid scheme, iterating over valid tuples to find an allowed one, is a universal technique for all kinds of constraints. The MGAC algorithm, maintaining generalized arc consistency during backtracking search, is the most popular technique to solve hard CSPs. It builds up a search tree from level 0 to level n, where n is the number of variables. At each node of the search tree, a variable x and a value a in $dom(x)$ are selected and a GAC algorithm is used to propagate the assignment. At level 0, GAC is usually enforced

to preprocess the problem before searching starts. A dead-end is reached if the propagation fails, then a backtracking occurs.

The classical GAC3rm algorithm is recalled here. We present the constraint-oriented version of GAC3rm in algorithm 1 and algorithm 2. The *changedVars* stores all the variables in *scp*(c) whose domains are changed during current invocation of algorithm 2. The *gacValues*(x) records the values in *dom*(x), which have already found a support. Residue supports are used at line 6 in algorithm 2 and multi-directional residue is implemented at lines 14 to 16. The *residue*(x, a, c) is a residue support for (x, a) on c, which was found as the support for (x, a). If *residue*(x, a, c) is valid, (x ,a) still has a support on c; otherwise, the FINDSUPPORT procedure is called to find a new support for (x, a). FINDSUPPORT iterates over all valid tuples involving (x, a) on c and check whether they are allowed by c. If an allowed one is found, it returns the tuple as the new support; otherwise, it returns *NULL*.

Algorithm 1 GAC3

```
1:  initialize Q;
2:  while Q is not empty do
3:      c ← the first constraint in Q;
4:      remove c from Q;
5:      changedVars ← REVISE(c);
6:      if changedVars = FAIL then
7:          return FAIL;
8:      for each variable x in changedVars do
9:          add all constraints involving x except c into Q;
10: return SUCCESS;
```

Algorithm 2 GAC3RM(c:constraint)

```
1:  for each variable x ∈ scp(c) and x is not assigned do
2:      gacValues(x) ← ∅;
3:  changedVars ← ∅;
4:  for each variable x ∈ scp(c) and x is not assigned do
5:      for each value a ∈ dom(x) \ gacValues(x) do
6:          if residue(x, a, c) is invalid then
7:              τ ← FINDSUPPORT(x, a, c);
8:              if τ = NULL then
9:                  remove a from dom(x);
10:                 if dom(x) = ∅ then
11:                     return FAIL;
12:                 changedVars ← changedVars ⋃ {x};
13:             else
14:                 for each x' ∈ scp(c) and x' is not
    assigned do
15:                     gacValues(x') ← gacValues(x') ⋃
    {τ[x']};
16:                     residue(x', τ[x'], c) ← τ;
17: return changedVars;
```

3 Growing tabular reduction: saving constraint checks during search

Before introducing the GTR algorithm, we give an example of positive repeats and negative repeats in MGAC3rm. Given a constraint c, x∈*scp*(c) and a∈*dom*(x), τ_1, τ_2, τ_3, τ_4, and τ_5 are tuples involving (x, a) on *scp*(c), where τ_3 and τ_5 satisfy c. At the beginning, all the five tuples are valid. After checking τ_1, τ_2, and τ_3, GAC3rm finds τ_3 as the support for (x, a) and records it as the residue support. (x, a) has a support as long as τ_3 is still valid. Later in the search, assuming τ_3 loses its validity due to some assignment and τ_1, τ_2, τ_4, and τ_5 are still valid, so during the search for another support, τ_1 and τ_2 are checked again. These constraint checks on τ_1 and τ_2 are negative repeats. After searching, τ_5 is found as a new support and it is recorded as new residue support for (x, a). The old residue support τ_3 is discarded. Later in the search, a backtracking occurs, assuming τ_5 loses its validity and τ_3 becomes valid again. Now, GAC3rm will seek a new support for (x, a) because τ_5 is no longer valid. It will check all valid tuples, so τ_3 is checked again. This constraint check on τ_3 is a positive repeat. Before this positive repeat on τ_3, if τ_1 and τ_2 are still valid, then there are another two negative repeats. If we did not discard the old residue support τ_3 when it became invalid and restore it after it becomes valid again, then this positive repeat will be avoided and the corresponding two negative repeats are also avoided.

In this section, we propose a new coarse-grained GAC algorithm avoiding all positive repeats, named growing tabular reduction (GTR). The GTR algorithm maintains a dynamic list of the tuples for each constraint c, which have been checked to be allowed by c. The algorithm contains two parts. In the first part, it iterates over all the recorded tuples in the list and check their validities. The values appearing in a valid one have supports on c. In the second part, it uses constraint checks to search for supports for only those values that have not found a support. This part is similar to what GAC3 does. If a new tuple is found as a new support of a value, it will be added into the dynamic list. Obviously, it may execute a great number of validity checks if we use this simple strategy during search. To solve this problem, we propose a method to avoid redundant validity checks. For each constraint, it records the tuples deleted at each level and restores them after a backtracking occurs. This method ensures that the tuples that have been verified to be invalid at level i will not be rechecked at level j (j>i). If a backtracking occurs, restoring tuples for each constraint can be done in constant time. In the following data structures, the tuples mean only the tuples that have been checked to satisfy constraint c, not all tuples satisfying c.

- *tupleList*(c) is a dynamic array of tuples. It records all the tuples that have been checked to be allowed by c. New tuples will be added at the end of the array. The

tuples in *tupleList*(c), which have not been verified to be invalid, are called active tuples.

- *firstActive*(c) is the position of the first active tuple in *tupleList*(c). It is initialized to 0. All active tuples in *tupleList*(c) are indexed from *tupleList*(c).*length*-1 to *firstActive*(c). The tuples before *firstActive*(c) are deleted.
- *levelLast*(c) is an array of size $n+1$ where n is the number of variables. *levelLast*(c)[p] is the position of the last invalid tuple of *tupleList*(c) removed when the search was at level p. Level 0 corresponds to the preprocessing step. *levelLast*(c)[p] $= -1$ if no tuple was removed at level p. It is used to record and restore *firstActive*(c) at each level during search.

The GTR algorithm for a constraint is present in algorithm 3 and it is called at line 5 in algorithm 1. When MGTR is propagating at level i, all the tuples in *tupleList*(c) before *firstActive*(c) have been verified to be invalid at previous levels, so they will not be checked. The first part of GTR is implemented at lines 3 to 11 and the second part is from lines 13 to 24. In part1, all the values with residue supports are identified. In part2, a GAC3 scheme is employed to find a support for the rest values. If a support is found, it is added to the end of *tupleList*(c) at line 22. If lines 3 to 11 and lines 22 to 24 are removed, algorithm 3 degenerates into a constraint-oriented GAC3 algorithm.

Algorithm 3 GTR(c:constraint, *level*:integer)

```
1:  for each variable x ∈ scp(c) and x is not assigned do
2:      gacValues(x) ← ∅;
3:  index ← tupleList(c).length-1;
4:  while index ≥ firstActive(c) do
5:      τ ← tupleList(c)[index];
6:      if τ is valid then
7:          for each x ∈ scp(c) and x is not assigned do
8:              gacValues(x) ← gacValues(x) ⋃ {τ[x]};
9:          index ← index -1;
10:     else
11:         REMOVETUPLE(index, level, c);
12: changedVars ← ∅;
13: for each variable x ∈ scp(c) and x is not assigned do
14:     for each value a ∈ dom(x)\gacValues(x) do
15:         τ ← FINDSUPPORT(x, a, c);
16:         if τ = NULL then
17:             remove a from dom(x);
18:             if dom(x) = ∅ then
19:                 return FAIL;
20:             changedVars ← changedVars ⋃ {x};
21:         else
22:             add τ to the end of tupleList(c);
23:             for each x ∈ scp(c) and x is not assigned do
24:                 gacValues(x) ← gacValues(x) ⋃ {τ[x]};
25: return changedVars;
```

Algorithm 4 removes the tuple indexed at i by switching it with the first active tuple indexed by *firstActive*(c) and increasing *firstActive*(c) by 1. In this way, all the deleted tuples are moved to the position before *firstActive*(c), so they will not be rechecked if no backtracking occurs. If a backtracking occurs at level i, the RESTORE procedure in algorithm 5 restores the tuples deleted at level i by simply recovering *firstActive*(c). After the restoring, the order of the active tuples may be different. This is not a problem, because all tuples after *firstActive*(c) will be checked in next invocation. This mechanism also ensures that the new added tuples will not be missed.

Algorithm 4 REMOVETUPLE(i, p:integer, c:constraint)

```
1: if levelLast(c)[p] = -1 then
2:     levelLast(c)[p] ← firstActive(c);
3: tempTuple ← tupleList(c)[i];
4: tupleList(c)[i] ← tupleList(c)[firstActive(c)];
5: tupleList(c)[firstActive(c)] ← tempTuple;
6: firstActive(c) ← firstActive(c) + 1;
```

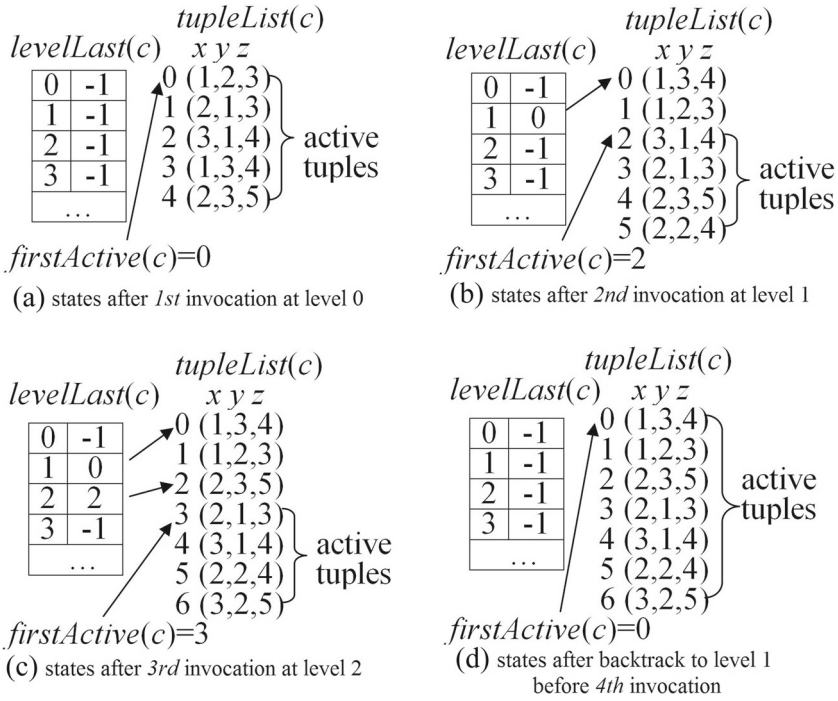
Algorithm 5 RESTORE(p:integer, c:constraint)

```
1: if levelLast(c)[p] ≠ -1 then
2:     firstActive(c) ← levelLast(c)[p];
3:     levelLast(c)[p] ← -1;
```

The example in Fig. 1 illustrates how the data structures of GTR work. A constraint c defined by a predicate $x+y=z$, $dom(x) = dom(y) = \{1, 2, 3\}$ and $dom(z) = \{3, 4, 5\}$. At the beginning, *tupleList*(c) is empty, *firstActive*(c) is 0 and the elements in *levelLast*(c) are -1.

- (a) The first invocation skips part1. It finds the supports for all values and adds the tuples into *tupleList*(c) at part2.
- (b) At level 1, (x, 1) is removed by other constraint. At part1, the algorithm checks tuples indexed from 4 to 0 and (1,2,3),(1,3,4) are removed. At part2, (2,2,4) is found as the support for (y, 2) and it is added to the end of *tupleList*(c).
- (c) At level 2, (y, 3) is removed. At part1, the algorithm checks tuples indexed from 5 to 2 and (2,3,5) is removed. As the new support for (z, 5), (3,2,5) is added to the end.
- (d) Assuming that c is never checked again until the searching backtracks to level 1. After the backtracking, both (x, 1) and (y, 3) are restored. In order to restore the deleted tuples, *firstActive*(c) is set to *levelLast*(c)[1] in Fig. 1b. Then, *levelLast*(c)[1] is set to -1. In the 4th invocation, all the 7 tuples will be checked at part1.

Fig. 1 Data structures of GTR



Proposition 1 *The worst-case time complexity of GTR to establish GAC at the preprocessing step is* $O(er^3d^{r+1})$ *with space complexity* $O(er^2d)$.

Proof At the preprocessing step, the tuples in *tupleList*(*c*) which are verified to be invalid can be discarded. In the worst case, each tuple in *tupleList*(*c*) supports only one value, so there are at most *rd* tuples in *tupleList*(*c*) at the preprocessing step. Therefore, the part1 of algorithm 3 costs r^2d time, because each validity check costs *r* time. The part2 cost r^2d^r time, because there are *r* variables in *scp*(*c*), *d* values in the domain of each variable, FINDSUPPORT(*x*, *a*, *c*) may iterate over at most d^{r-1} tuples and each constraint check costs *r* time. So the worst-case time complexity of algorithm 3 is $O(r^2d^r)$. For each constraint *c*, algorithm 3 will be called at most *rd* times and there are *e* constraints; therefore, the worst-case time complexity to establish GAC is $O(er^3d^{r+1})$. As for space, there are at most *rd* tuples in *tupleList*(*c*), each tuple needs *r* space and there are *e* constraints, so the worst-case space complexity is $O(er^2d)$. □

Proposition 2 *The worst-case space complexity of maintaining GTR during search is* $O(erd^r)$.

Proposition 2 is straightforward, because the total number of tuples recorded in *tupleList*(*c*) is at most d^r when GTR is maintained during search.

Proposition 3 *The GTR algorithm avoids all positive repeats when it is maintained during search.*

Proof For each constraint *c*, all the tuples that are checked to be allowed are stored in *tupleList*(*c*), so a positive repeat occurs iff a tuple in *tupleList*(*c*) is checked. During each invocation, all the tuples before *firstActive*(*c*) are invalid, so they will not be checked. The tuples after *firstActive*(*c*) are valid and all the values appearing in these tuples are identified as having supports. In part2, the algorithm seeks supports for only those values that have not been identified as having supports and each of the tuples after *firstActive*(*c*) contains at least one value that has been identified as having supports, so these tuples will not be checked. Therefore, no positive repeat occurs when maintaining GTR during search.

Property 1 *For each constraint c, there is no duplicate tuple in tupleList(c).*

The property is true, because maintaining GTR during search has no positive repeat, so a tuple allowed by *c* will be checked and added into *tupleList*(c) at most once. □

Besides naive GTR, the algorithm can be improved by the methods used in STR2. The improved version, GTR2, is shown in algorithm 6. GTR2 does not improve the worst-case time complexity, but it avoids some unnecessary operations. The additional data structures used in GTR2 is same as those used in STR2. The S^{sup} records the variables that at least one value has not found a support. If all values in *dom*(*x*) have already found a support, we remove *x* from S^{sup}, then efficiency is gained by iterating over only variables in S^{sup} at lines 15, 24, and 34. The S^{val} records

the variables whose domain was changed between last invocation and this invocation. The $lastSize(c)[x]$ records the domain size of x after each invocation of constraint c and is used to determine whether the domain of a variable was changed recently. To check the validities of the tuples, we do not check the values of variable x if $dom(x)$ was not changed between last invocation and this invocation. Therefore, at line 14 of algorithm 6, the procedure is Valid(τ, S^{val}) checks only the variables in S^{val} to verify if τ is valid.

Algorithm 6 GTR2(*c*:constraint, *level*:integer)

```
1:  S^sup ← ∅;
2:  S^val ← ∅;
3:  if the last assigned variable x_l ∈ scp(c) then
4:      S^val ← S^val ⋃ {x_l};
5:  for each variable x ∈ scp(c) and x is not assigned do
6:      gacValues(x) ← ∅;
7:      S^sup ← S^sup ⋃ {x};
8:      if |dom(x)| ≠ lastSize(c)[x] then
9:          S^val ← S^val ⋃ {x};
10:         lastSize(c)[x] ← |dom(x)|;
11: index ← tupleList(c).length-1;
12: while index ≥ firstActive(c) do
13:     τ ← tupleList(c)[index];
14:     if isValid(τ, S^val) then
15:         for each x ∈ S^sup do
16:             if τ[x] ∉ gacValues(x) then
17:                 gacValues(x) ← gacValues(x) ⋃ {τ[x]};
18:                 if |gacValues(x)| = |dom(x)| then
19:                     S^sup ← S^sup \ {x};
20:         index ← index -1;
21:     else
22:         REMOVETUPLE(index, level, c);
23: changedVars ← ∅;
24: for each variable x ∈ S^sup do
25:     for each value a ∈ dom(x)\gacValues(x) do
26:         τ ← FINDSUPPORT(x, a, c);
27:         if τ = NULL then
28:             remove a from dom(x);
29:             if dom(x) = ∅ then
30:                 return FAIL;
31:             changedVars ← changedVars ⋃ {x};
32:         else
33:             add τ to the end of tupleList(c);
34:             for each x ∈ S^sup do
35:                 if τ[x] ∉ gacValues(x) then
36:                     gacValues(x) ← gacValues(x) ⋃
    {τ[x]};
37:                     if |gacValues(x)| = |dom(x)| then
38:                         S^sup ← S^sup \ {x};
39:     lastSize(c)[x] ← |dom(x)|;
40: return changedVars;
```

4 Discussion and related works

GTR avoids all positive repeats and the corresponding negative repeats executed before each positive repeat. However, the negative repeats for proving a value having no support cannot be avoided by GTR. When GTR is maintained during search, the part1 iterates over $O((1\text{-}t)d^r)$ allowed tuples where t is constraint tightness. For each value, the part2 tries $O(td^{r-1})$ disallowed tuples before finding a support. If the constraint is tight, $(1\text{-}t)d^r$ is a relatively small number and it is relatively harder to find a support for a value; therefore, GTR should be efficient when being used on tight constraints. On the contrary, if the constraint is loose, it is relatively easier to find a support and $(1\text{-}t)d^r$ may be a large number, so GTR may be inefficient for loose constraints.

The part1 of GTR is similar to the simple tabular reduction (STR) algorithm [8, 13, 19, 21] which operates on extensional constraints listing all allowed tuples in a table. Both them identify the values which have supports by iterating over the allowed tuples. One difference is that STR iterates over a static table of allowed tuples of a constraint, whereas GTR iterates over a dynamic list of tuples that are checked to be allowed. When a backtracking occurs, STR has a mechanism to restore tuples with few time cost, which is suitable for only static tables. GTR employs a similar mechanism to restore deleted tuples with the same time cost, which works on dynamic lists. Another difference is that STR determines if a value has supports after iterating the table, because all allowed tuples are already in the table, whereas GTR usually identifies some of the values with supports after iterating the dynamic list, for the rest values, it uses constraint checks to determine if they have supports. We prefer STR to GTR on extensional constraints, because STR is a specialised algorithm for these constraints. However, GTR is a generic GAC algorithm that works on all kinds of constraints.

GTR is also similar to the GIC4 algorithm proposed in [2], but they enforce different consistencies. Another difference between them is that GTR records the results of constraint checks for each constraint, whereas GIC4 maintains only one list of global solutions. GIC4 enforces global consistency, so maintaining GIC4 during search in backtrack-free and it does not restore deleted solutions. The major advancement of GTR over GIC4 is that GTR has a mechanism to cope with backtracking and the deleted tuples can be restored with low cost.

Exploring multiple residue supports, the GAC3rm_*k* algorithms [11], where k is the maximum number of residues, enforces less positive repeats than GAC3rm. When searching for a support for a value, GAC3rm_*k* first checks the validities of the k residues. If none of the residues is valid, it starts to search for a new support by constraint checks. If k is set to infinity, no residue support is discarded,

Table 1 Results on non-binary instances with tight constraints

Instance		GAC3rm	GAC3rm2	GAC3rm_inf	GTR	GTR2	GAC3.1	GAC3.2
BIBD (30)	cpu	23.62	20.21	87.51 (1 out)	15.03	*13.27*	106.86 (1 out)	22.31
	cc	278M	224M	–	127M	127M	–	203M
	vc	25M	33M	–	42M	42M	–	28M
	rn	11K	21K	–	119K	119K	–	20K
BIBD 6-30-15-3-6	cpu	149	132	298	53	*48*	1059	117
	cc	2.4B	2.1B	0.8B	0.8B	0.8B	13.7B	1.6B
	vc	20M	25M	10B	40M	40M	20M	23M
	rn	4K	8K	212K	212K	212K	4K	8K
BIBD 7-42-18-3-6	cpu	208	198	236	161	*156*	out	177
	cc	2.4B	2.3B	1.9B	1.9B	1.9B	–	1.7B
	vc	6.7M	8M	1.6B	13M	13M	–	7.5M
	rn	8K	15K	100K	100K	100K	–	14K
Costas Array (8)	cpu	25.29	25.03	37.12	23.14	*21.92*	29.54	29.47
	cc	540M	429M	300M	300M	300M	341M	458M
	vc	198M	298M	1.5B	260M	260M	199M	221M
	rn	4.7K	9.2K	26.2K	26.2K	26.2K	4.7K	7.2K
Costas Array-16	cpu	33.15	32.92	43.35	29.12	*27.65*	38.02	38.25
	cc	711M	564M	391M	391M	391M	453M	605M
	vc	261M	392M	1.8B	312M	312M	261M	291M
	rn	7.3K	14.5K	50.7K	50.7K	50.7K	7.3K	11.2K
Costas Array-17	cpu	152	150	231	140	*132.75*	178	177
	cc	3.2B	2.6B	1.8B	1.8B	1.8B	2.1B	2.8B
	vc	1.2B	1.8B	9.4B	1.6B	1.6B	1.2B	1.3B
	rn	8.8K	17.5K	79K	79K	79K	8.8K	13.5K
radar 8-30-3-0-15	cpu	0.75	0.69	0.67	0.60	*0.55*	0.83	0.71
	cc	2.2M	2.0M	1.6M	1.6M	1.6M	2.5M	2.0M
	vc	244K	357K	10M	787K	787K	243K	326K
	rn	2.6K	5K	18K	18K	18K	2.6K	4.8K
mknap-1-3	cpu	0.291	*0.279*	0.386	0.30	0.281	0.406	0.343
	cc	1.11M	1.10M	1.09M	1.09M	1.09M	1.26M	1.08M
	vc	49.6K	65K	732K	50K	50K	49.4K	59K
	rn	438	866	8K	8K	8K	438	670
mknap-1-4	cpu	235	218	453	180	*171*	372	209
	cc	721M	708M	543M	543M	543M	917M	516M
	vc	9.5M	13M	28B	29M	29M	9.5M	11M
	rn	614	1218	591K	591K	591K	614	941
magic-Square-5	cpu	3.45	3.06	5.98	2.10	*2.01*	3.0	2.32
	cc	194M	163M	81M	81M	81M	135M	112M
	vc	4.3M	6.3M	293M	17M	17M	4.3M	5.7M
	rn	1.5K	2.99K	112K	112K	112K	1.5K	2.79K
magic-Square-6	cpu	17.90	16.72	14.98	13.46	*13.30*	28.39	13.76
	cc	1.0B	0.95B	0.77B	0.77B	0.77B	1.5B	0.76B
	vc	1.5M	2.1M	26M	5.0M	5.0M	1.5M	1.9M
	rn	3.0K	6.0K	96K	96K	96K	3.0K	5.7K
magic- Square-7	cpu	out	out	1066	1062	*1026*	out	out
	cc	–	–	57B	57B	57B	–	–
	vc	–	–	162M	25M	25M	–	–
	rn	–	–	357K	357K	357K	–	–

The best cpu time is in italic. rn is the number of residue supports

so GAC3rm_*infinity* also avoids all positive repeats and the worst-case space complexity of GAC3rm_*infinity* is the same as that of GTR. However, compared with GTR, GAC3rm_*infinity* cannot avoid redundant validity checks. It may execute much more validity checks than GTR. The dynamic list of GTR may be adopted in MGAC3rm_*infinity* to avoid redundant validity checks, but each value on each constraint needs a dynamic list, so it maintains *rd* dynamic lists for each constraint, which is much more than that of GTR.

The MAC3cache algorithm [14] caches the results of all possible constraint checks. This method was proposed to work on binary CSPs, because caching the results of all possible constraint checks will cost d^r space for each r-arity constraint. When a constraint check is repeated, MAC3cache can get the result with low cost. GTR also caches the results of some constraint checks, but only the allowed results, not all possible constraint checks. Actually, we would prefer the STR algorithm to a classical GAC algorithm after all allowed tuples are cached.

5 Experiments

The experiments were run on a PC with Intel(R) Core(TM) i5-3210M CPU @2.5GHz, 4GB RAM, JDK 1.7. The performance of maintaining each GAC (or AC for binary instances) algorithm for finding the first solution or proving unsatisfiable is measured by CPU time (cpu) in seconds, number of constraint checks (cc), and number of validity checks (vc). The numbers of cc and vc are present by kilo (K), million (M), and billion (B). Timeout (out) is 1200 s and maximum memory is set to 1000M. In the average results, the cpu time of timeout instances are counted as 1200 s. The results of those instances where all solvers are timeout are eliminated from the average results. The variable ordering heuristic is *dom/wdeg* [6] equipped with a random restart strategy [7, 22]. The GAC3rm_*k* algorithms are implemented with a static FIFO policy and the later added residues are checked earlier.

Firstly, we compared GTR with GAC3rm, GAC3rm_2, GAC3rm_*infinity*, GAC3.1, and GAC3.2 on some non-binary

Table 2 Results on non-binary instances without tight constraints

Instance		GAC3rm	GAC3rm2	GAC3rm_inf	GTR	GTR2	GAC3.1	GAC3.2
Chessboard coloration 25-25-2	cpu	15.44	15.41	*13.96*	18.69	41.32	–	–
	cc	12M	2.0M	0.6M	0.6M	0.6M	out	out
	vc	34.6M	48.7M	63.7M	43.2M	43.2M	of	of
	rn	0.7M	1.2M	0.5M	0.5M	0.5M	memory	memory
Chessboard Coloration 9-9-3	cpu	*1.76*	2.03	4.26	2.49	3.26	4.46	3.62
	cc	8.1M	4.5M	0.2M	0.2M	0.2M	10M	6.2M
	vc	24.8M	35.3M	149M	54.4M	54.4M	24.8M	31.5M
	rn	15K	31K	85K	85K	85K	15K	28K
tdsp C1-1-20	cpu	4.42	4.40	16.0	14.91	11.40	5.66	*4.15*
	cc	26M	22M	3M	3M	3M	34M	22M
	vc	30M	44M	1.4B	675M	675M	30M	41M
	rn	10K	18K	409K	409K	409K	10K	18K
tdsp C1-1-21	cpu	5.60	5.30	17.21	13.57	10.22	6.29	*5.07*
	cc	33M	26M	7M	7M	7M	34M	26M
	vc	37M	54M	1.4B	522M	522M	37M	51M
	rn	10K	19K	392K	392K	392K	10K	19K
Golomb ruler 34-9-a4	cpu	*6.58*	10.06	190	88	81	14.5	15.8
	cc	50M	44M	8M	8M	8M	67M	47M
	vc	132M	202M	9.1B	1.9B	1.9B	135M	186M
	rn	63K	117K	7M	7M	7M	63K	124K
Golomb Ruler 44-10-a4	cpu	*51*	95	–	–	–	146	162
	cc	422M	373M	out	out	out	548M	399M
	vc	1.02B	1.58B	of	of	of	1.04B	1.47B
	rn	132K	248K	memory	memory	memory	132K	261K

The best cpu time is in italic

instances containing intensional constraints. GTR and the GAC3rm algorithms are light when being used in search, whereas GTR2 maintains the data structure *lastSize*. GAC3.1 and GAC3.2 maintain the data structure *last* during search. As we mentioned at Section 4, the gain in efficiency that GTR brings is related to constraint tightness, so we tested five problems contain some tight constraints (*Multi-Knapsack Instances* (*mknap*), *magicSquare*, *Balanced Incomplete Block Designs* (*BIBD*), *CostasArray*, *Radar Surveillance* (*radar*)) and three problems contain only loose constraints (*ChessboardColoration*, *Two-Dimensional Strip Packing Problems* (*tdsp*) and *Golomb Ruler*). All these benchmark instances are downloaded from http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html.

Table 1 presents the results of some problems containing tight constraints. The integers in the brackets under instance names are the number of tested instances in that series. On the *mknap*, *magicSquare*, and *BIBD* instances, we can see that GTR algorithms trade a relatively small number of validity checks (compared to the number of constraint checks) for some constraint checks. Consequently, they save some cpu time. More specifically, the gain on each *mknap* instance is from the only tight constraint and the gains on *magicSquare* and *BIBD* instances are from the *weightedSum* constraints in these instances. On the *CostasArray* instances, GTR algorithms save a little time from the trading, it reduces the sum of validity checks and constraint checks. The *radar-8-30-3-0-15* instance is the only representative which needs 729 search tree nodes to solve (others are easier). GAC3.2 enforces less constraint checks than GTR on the *mknap* instances, because it maintains the data structure *last* during search, which avoids some negative repeats.

Table 2 presents the results of some problems containing only loose constraints. The results of *Chessboardcoloration*, *tdsp*, and *Golomb ruler* show that it is not recommended to trade constraint checks for validity checks on loose constraints. The GTR algorithms are not efficient on these instances, but they are still better than GAC3rm_*infinity*, because GAC3rm_*infinity* needs much more validity checks. The GTR2 algorithm maintains additional data structures during search, so it costs more cpu time than the GTR algorithm on the instances containing a large number of constraints, e.g., the *Chessboardcoloration* instances, but it improves GTR in general.

Secondly, we compared GTR with AC3rm on binary instances including real-world, patterned, and academic binary instances. The results in Table 3 show that GTR is not efficient on these binary instances even on those instances containing tight constraints [12]. The main reason is that GTR is built in a GAC scheme which is less efficient than an AC scheme. Both GTR and an AC scheme check the validity of residue supports. The GTR needs two operations to check the validity on binary constraints, whereas an AC scheme needs only one operation. Each valid residue support supports only one value in both schemes. For an r-ary constraint, GTR needs r operations to check the validity of a residue support and the residue support supports *r* values.

Table 3 Results on binary real-world, patterned, and academic problems

Instance		AC3rm	AC3rm2	GTR
RLFAP	cpu	27.1	29.4	172 (1 out)
scens11 (10)	cc	307M	215M	–
RLFAP	cpu	40.1	42.9	350
scens11-f4	cc	486M	334M	141M
RLFAP	cpu	2.97	3.13	23.7
scens11-f6	cc	36M	26M	11M
Job-Shop	cpu	84.56	83.75	126 (4 out)
(41)	cc	2.4B	1.9B	–
Job-Shop	cpu	13.8	14.7	114
e0ddr1-10-by-5-1	cc	400M	371M	328M
Job-Shop	cpu	1069	991	out
e0ddr2-10-by-5-10	cc	33B	23B	–
Queens-Knights	cpu	1.05	1.05	8.83
(12)	cc	10.3M	9.9M	8.7M
Queens-Knights	cpu	3.52	3.54	31.23
25-5-add	cc	35M	34M	30M
Queens-Knights	cpu	4.12	4.21	46.97
25-5-mul	cc	41M	39.8M	34.9M

In summary, the GTR algorithms have few improvement on binary instances, but they bring some improvements on non-binary tight constraints.

6 Conclusion and future work

In this paper, we propose a new generalized arc consistency algorithms GTR and its improved version GTR2, which avoid all positive repeats. The experimental results show that the GTR algorithms save a number of constraint checks. It is not suggested to use GTR on binary instances, but it works well on those tight constraints with larger arity. If we adopt the dynamic list of supports in some higher level local consistencies where the cost of finding a support is much more expensive, potentially, it may bring more improvement on cpu time.

Acknowledgments This work was supported by the Fundamental Research Funds for the Central Universities (NO. 2412016KJ034), the Education Department of Jilin Province (Project NO. JJKH20170911KJ) and Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University (NO.93K172017K06).

Compliance with ethical standards

Conflict of interests The authors declare that they have no conflicts of interest.

References

1. Bessière C (1994) Arc-consistency and arc-consistency again. Artif Intell 65:179–190
2. Bessière C, Fargier H, Lecoutre C (2013) Global inverse consistency for interactive constraint satisfaction. In: Proceedings of CP'13, pp 159–174
3. Bessière C, Freuder EC, Régin JC (1999) Using constraint metaknowledge to reduce arc consistency computation. Artif Intell 107:125–148
4. Bessière C, Régin JC (1997) Arc consistency for general constraint networks: preliminary results. In: Proceedings of IJCAI'97, pp 398–404
5. Bessière C, Régin JC, Yap R, Zhang Y (2005) An optimal coarse-grained arc consistency algorithm. Artif Intell 165:165–185
6. Boussemart F, Hemery F, Lecoutre C, Sais L (2004) Boosting systematic search by weighting constraints. In: Proceedings of ECAI'04, pp 146–150
7. Gomes C, Selman B, Kautz H (1998) Boosting combinatorial search through randomization. In: Proceedings of AAAI'98, pp 431–437
8. Lecoutre C (2011) STR2: optimized simple tabular reduction for table constraints. Constraints 16:341–371
9. Lecoutre C, Boussemart F, Hemery F (2003) Exploiting multidirectionality in coarsegrained arc consistency algorithms. In: Proceedings of CP'03, pp 480–494
10. Lecoutre C, Hemery F (2007) A study of residual supports in arc consistency. In: Proceedings of IJCAI'07, pp 125–130
11. Lecoutre C, Likitvivatanavong C, Shannon S, Yap R, Zhang Y (2008) Maintaining arc consistency with multiple residues. Constraint Program Lett 2:3–19
12. Li H (2017) Narrowing support searching range in maintaining arc consistency for solving constraint satisfaction problems. IEEE access. doi:10.1109/ACCESS.2017.2690672
13. Li H, Liang Y, Guo J, Li Z (2013) Making simple tabular reduction works on negative table constraints. In: Proceedings of AAAI'13, pp 1629–1630
14. Likitvivatanavong C, Zhang Y, Bowen J, Freuder EC (2004) Arc consistency in MAC a new perspective. In: Proceedings of CPAI'04 workshop held with CP'04, pp 93–107
15. Likitvivatanavong C, Zhang Y, Shannon C, Bowen J, Freuder EC (2007) Arc consistency during search. In: Proceedings of IJCAI'07, pp 137–142
16. Mackworth AK (1977) Consistency in networks of relations. Artif Intell 8:99–118
17. Mohr R, Henderson TC (1986) Arc and path consistency revisited. Artif Intell 28:225–233
18. Sabin D, Freuder EC (1994) Contradicting conventional wisdom in constraint satisfaction. In: Proceedings of ECAI'94, pp 125–129
19. Ullmann JR (2007) Partition search for non-binary constraint satisfaction. Inf Sci 177:3639–3678
20. van Dongen MRC (2004) Saving support-checks does not always save time. Artif Intell Rev 21:317–334
21. Wang R, Xia W, Yap R, Li Z (2016) Optimizing simple table reduction with bitwise representation. In: Proceedings of IJCAI'16, pp 787–793
22. Walsh T (1999) Search in a small world. In: Proceedings of IJCAI'99, pp 1172–1177