IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received February 12, 2017, accepted March 10, 2017, date of publication April 4, 2017, date of current version May 17, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2690672

Narrowing Support Searching Range in Maintaining Arc Consistency for Solving Constraint Satisfaction Problems

HONGBO LI
School of Computer Science and Information Technology, Northeast Normal University, Changchun 130117, China
Corresponding author: H. Li (lihb905@nenu.edu.cn)

This work was supported in part by the Fundamental Research Funds for the Central Universities under Grant 2412016KJ034, in part by the Education Department of Jilin Province under Project JJKH20170911KJ, and in part by the Key Laboratory of Symbolic Computation and Knowledge Engineering, Jilin University, Ministry of Education of China.

ABSTRACT Arc consistency is the most popular filtering technique for solving constraint satisfaction problems. Constraint check plays a central role in establishing arc consistency. In this paper, we propose a method to save constraint checks in maintaining coarse-grained arc consistency during backtracking search for solving the constraint satisfaction problems. We reduce the support searching range by utilizing the information generated by an AC3.1 algorithm at preprocessing step. Compared with the existing maintaining arc consistency (MAC) algorithms, the proposed MAC3$_{be}$ algorithm saves constraint checks without maintaining additional data structures at each search tree node. Our experimental results show that MAC3$_{be}$ saves both constraint checks and CPU time while solving some benchmark problems.

INDEX TERMS Constraint satisfaction problem, constraint propagation, arc consistency, residue support.

I. INTRODUCTION

Constraint satisfaction problems (CSPs) have been widely used in artificial intelligence, operations research and other areas of computer science. Finding a solution in a CSP is NP-hard. Backtracking search is usually used to solve CSPs. Exploring the whole space of instantiations is of course too expensive, so some filtering techniques pruning values from domains to reduce search space are integrated in backtracking search, such as arc consistency (AC), max restricted path consistency (maxRPC) [2]–[4], restricted path consistency (RPC) [5]. Besides, filtering techniques have also been used in quantified CSP [6]–[9] which is a generalization of the CSP that can be used to model combinatorial problems containing contingency or uncertainty.

AC is the most popular filtering technique used in solving constraint satisfaction problems. Maintaining arc consistency algorithm (MAC) [10], [11], maintains AC during backtracking search, is the most popular technique to solve large and hard CSPs. A number of AC algorithms have been proposed in the past 30 years. These algorithms are classified into the fine-grained and the coarse-grained. The former [12]–[14] are based on a value-oriented propagation scheme and the latter [1], [15]–[17] are based on a constraint-oriented propagation scheme. Constraint check plays a central role in establishing arc consistency. The fine-grained algorithms use more elaborate data structures to avoid useless constraint checks. These data structures need to be maintained during search, in other words, they need to be stored and restored at each search tree node in order to cope with backtracking. It has been recognized that an efficient MAC algorithm usually has two features: (1) the AC algorithm it uses is efficient, (2) it maintains few data structure during search. We use MAC$_N$ to denote a MAC algorithm uses AC$_N$ algorithm. The AC3 algorithms are universal coarse-grained algorithms. The AC3 family usually maintain few data structure during search, so they are more popular when being used in MAC. The original AC3 [1] algorithm has the worst-case time complexity $O(ed^3)$ where e is the number of constraints and d is the maximum variable domain size. By recording *last* supports, the AC3.1 [15] algorithm avoids some repeated constraint checks and has an optimal worst-case time complexity $O(ed^2)$. However, MAC3.1 is inefficient due to maintaining its additional data structure *last* during search. The AC3$_{rm}$ algorithm [17] improves AC3 by making use of multi-directional residue supports. MAC3$_{rm}$ maintains as few data structure as MAC3. It has been considered as the most efficient universal MAC algorithm. Although saving constraint checks does not always save time [18], it is still important to save constraint checks in MAC algorithms when the costs of constraint checks are relatively expensive. Besides these

2169-3536 © 2017 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

generic algorithms, some efficient AC algorithms designed for specific constraints have been developed, such as extensional constraints enumerating all tuples allowed or disallowed by the constraints [19]–[23].

In this paper, we propose a method to save constraint checks in coarse-grained MAC algorithms. The new algorithm, named MAC3$_{be}$, saves constraint checks without maintaining additional data structure. The idea is making use of the information generated by AC3.1 at preprocessing step to narrow the support searching range of the variable domains at each search tree node. We use AC3.1 to find the *beginning* support and *end* support for each value. The values before *beginning* support and after *end* support will not be checked, so the scopes of finding supports are reduced. The experiments were run on some random, academic, patterned and real-world instances. The results have shown that, compared with the classical MAC3$_{rm}$ and MAC3$_{rm2}$ algorithms, MAC3$_{be}$ saves a number of constraint checks. When the constraint checks are relatively expensive, it saves up to 3 times cpu time.

This paper is organized as follows. Section 2 provides some technical background about constraint satisfaction problem and AC3 algorithm frame. Section 3 recalls the details of AC3.1 and AC3$_{rm}$. The new algorithm is introduced in Section 4. The related works are discussed in Section 5. The experimental results and the analysis are in Section 6. Finally, Section 7 is conclusion.

II. BACKGROUND

A constraint satisfaction problem (CSP) P is a triple $P = \langle X, D, C\rangle$ where X is a set of n variables $X = \{x_1, x_2 \ldots x_n\}$, D is a set of domains $D = \{dom(x_1), dom(x_2) \ldots dom(x_n)\}$ where $dom(x_i)$ is a finite set of possible values for variable x_i, C is a set of e constraints $C = \{c_1, c_2 \ldots c_e\}$. A constraint c consists of two parts, an ordered set of variables $scp(c) = \{x_{i1}, x_{i2} \ldots x_{ir}\}$ and a subset of the Cartesian product $dom(x_{i1}) \times dom(x_{i2}) \times \ldots \times dom(x_{ir})$ that specifies the allowed (or disallowed) combinations of values for the variables $\{x_{i1}, x_{i2} \ldots x_{ir}\}$. An element of $dom(x_{i1}) \times dom(x_{i2}) \times \ldots \times dom(x_{ir})$ is called a tuple on $scp(c)$, denoted by τ. Verifying if a given tuple is allowed by a constraint is called a constraint check. $|scp(c)|$ is the arity of c. We focus on binary constraints in this work. c_{ij} denotes a constraint involving variables x_i and x_j. (x_i, a) denotes the value a for variable x_i. The pair of a variable x_i and a constraint c_{ij} involving x_i is called an arc, denoted by (x_i, c_{ij}). *nil* is defined as a value not belonging to any domain.

Definition 1 (Arc Consistency [1]): Given a CSP $P = \langle X, D, C\rangle$, an arc ($x_i$, c_{ij}) is consistent iff $\forall$ (x_i, a) $\in dom(x_i)$, there exists a value (x_j, b) $\in dom(x_j)$ such that (a, b) satisfies c_{ij}. P is arc consistent iff $\forall\ x_i \in X$, $dom(x_i)$ is not empty and every arc in P is consistent. (x_j, b) is called a support for (x_i, a) in constraint c_{ij}.

To establish arc consistency in a CSP, AC algorithms seek supports for every value (x_i, a) in the constraints involving x_i and remove those values without any support on these constraints. If the domain of a variable is empty, AC fails. The AC3 algorithms iterate over the domain of x_j to seek a support for a value (x_i, a) on constraint c_{ij}. The MAC algorithms build up a search tree from level 0 to level n, where n is the number of variables. At level 0, arc consistency is established in the problem for preprocessing. After that, at each node of the search tree, one variable x_i and a value a in $dom(x_i)$ are selected and an AC algorithm is used to propagate the assignment AC($x_i = a$). If the propagation fails, a dead-end is reached and a backtracking occurs, otherwise, MAC goes to next level.

The AC3 algorithm frame is recalled in Algorithm 1 and Algorithm 2. The propagation queue stores all the arcs that need to be revised. At the preprocessing phase, initializing Q adds all arcs in the problem into Q. At the searching phase, initializing Q adds only the arcs related to current assignment into Q. The algorithm removes and revises the arcs in Q one by one until Q is empty or a domain wipes out. The revise(x_i, c_{ij}) procedure removes every value without a support from $dom(x_i)$. If no value is removed, it returns false, otherwise true. In Algorithm 1, if the revise procedure removes some value, the affected arcs will be added into Q at lines 8 and 9. In this frame, the difference among AC3-based algorithms is how to seek support for values. If no support is found, the seekSupport procedures return *nil*, otherwise they return the support.

Algorithm 1 AC3

1: initialize Q;
2: **while** Q is not empty **do**
3:   select and remove an arc (x_i, c_{ij}) from Q;
4:   **if** revise(x_i, c_{ij}) **then**
5:     **if** $dom(x_i) = \emptyset$ **then**
6:       return *fail*;
7:     **else**
8:       **for** each constraint c_{ik} such that $k \neq j$ **do**
9:         $Q \leftarrow Q \bigcup (x_k, c_{ik})$;
10: return *success*;

Algorithm 2 Revise(x_i, c_{ij})

change $\leftarrow$ *false*;
for each value (x_i, a) $\in dom(x_i)$ **do**
  if seekSupport(x_i, a, c_{ij}) = *nil* **then**
    remove a from $dom(x_i)$;
    change $\leftarrow$ *true*;
return *change*;

III. THE EXISTING ALGORITHMS: AC3.1 AND AC3$_{rm}$

In this section, we recall the two algorithms, AC3.1 and AC3$_{rm}$, which are most related to our method. Both them are based on the AC3 frame. We introduce the details of the seeking support procedures of the two algorithms in Algorithm 3 and Algorithm 4 respectively.

AC3.1 associates each $dom(x_i)$ with a total ordering. According to the ordering, the function $head(x_i)$ returns the first value in $dom(x_i)$, $tail(x_i)$ returns the last value in $dom(x_i)$, $next(a, dom(x_i))$ returns the first value in $dom(x_i)$ that is after a. $next(tail(x_i), dom(x_i))$ returns nil and $next(nil, dom(x_i))$ returns $head(x_i)$.

A data structure $last(x_i, a, c_{ij})$ stores the last support of (x_i, a) in constraint c_{ij}. Each $last(x_i, a, c_{ij})$ is initialized to nil. When seeking a support for (x_i, a) in c_{ij}, if $last(x_i, a, c_{ij})$ is still in $dom(x_j)$, it returns the last support, otherwise it searches for a new support from $last(x_i, a, c_{ij})$ to $tail(x_i)$. If a new support (x_j, b) is found, $last(x_i, a, c_{ij})$ is updated by (x_j, b), otherwise, (x_i, a) is removed from $dom(x_i)$. The data structure $last(x_i, a, c_{ij})$ ensures that every value (x_j, c) before $last(x_i, a, c_{ij})$ is not a support of (x_i, a) in constraint c_{ij}, because either (x_j, c) has been removed from $dom(x_j)$ or (a, c) does not satisfy c_{ij}. Therefore, we have the following straight-forward property.

Property 1: After applying AC3.1 to a CSP P at preprocessing step, all the values before $last(x_i, a, c_{ij})$ in $dom(x_j)$ will never be a support of (x_i, a) in c_{ij}.

When maintaining AC3.1 during search, in order to cope with backtracking, every $last(x_i, a, c_{ij})$ needs to be recorded before each propagation and restored after each failure. Maintaining the data structure *last* at each node costs $O(ed)$ time. This reduces the efficiency of MAC3.1.

Algorithm 3 seekSupport3.1(x_i, a, c_{ij})

$b \leftarrow last(x_i, a, c_{ij})$;
if $b \notin dom(x_j)$ **then**
    $b \leftarrow next(b, dom(x_j))$;
    while $b \neq nil$ **do**
        if (a, b) satisfies c_{ij} **then**
            $last(x_i, a, c_{ij}) \leftarrow b$;
            return b;
        else
            $b \leftarrow next(b, dom(x_j))$;
return b;

The AC3$_{rm}$ algorithm uses a data structure $residue(x_i, a, c_{ij})$, initialized to nil, to record the most recent support of (x_i, a) in c_{ij}. The residue support technique was first introduced in [24]. Different from AC3.1, once the residue support has been removed, AC3$_{rm}$ searches for a new support from $head(x_i)$ to $tail(x_i)$. We can see that from line 3 of Algorithm 4. Therefore, $residue(x_i, a, c_{ij})$ does not need to be maintained during search. Besides residue support technique, AC3$_{rm}$ further explores multi-directional technique which is based on the fact that if (x_j, b) is a support of (x_i, a) in c_{ij}, then (x_i, a) is also a support of (x_j, b) in c_{ij}. The technique is implemented at lines 6 and 7 of Algorithm 4.

IV. THE NEW ALGORITHM: AC3$_{be}$

The time complexity of storing or restoring the data structure *last* at each search tree node is $O(ed)$. Maintaining the data structure makes MAC3.1 less efficient. An alternative way to this problem is not to record the *last* data structure before propagation and re-initialize each $last(x_i, a, c_{ij})$ to nil after each failure. This strategy reduces the cost of maintaining the data structure, but increases the number of constraint checks for seeking supports. Our aim is to reduce constraint checks without maintaining the data structure *last*.

Algorithm 4 seekSupport$_{rm}(x_i, a, c_{ij})$

1: **if** $residue(x_i, a, c_{ij}) \in dom(x_j)$ **then**
2:     return $residue(x_i, a, c_{ij})$;
3: $b \leftarrow head(x_j)$;
4: **while** $b \neq nil$ **do**
5:     **if** (a, b) satisfies c_{ij} **then**
6:         $residue(x_i, a, c_{ij}) \leftarrow b$;
7:         $residue(x_j, b, c_{ij}) \leftarrow a$;
8:         return b;
9:     **else**
10:         $b \leftarrow next(b, dom(x_j))$;
11: return b;

We introduce a strategy narrowing the support searching scope without maintaining additional data structures. AC3.1 is employed at preprocessing phase and after it is done, we record $last(x_i, a, c_{ij})$ in a new data structure $beginning(x_i, a, c_{ij})$. According to Property 1, every value before $beginning(x_i, a, c_{ij})$ will never be a support for (x_i, a) in c_{ij}, so if we do not use *last* during search, we can search for a new support from $beginning(x_i, a, c_{ij})$ to $tail(x_j)$ instead of searching from $head(x_j)$ to $tail(x_j)$. Note that the data structure *beginning* is copied from the data structure *last* and it is fixed after that, so it is not maintained during search.

The AC3.1 algorithm searches for a support in the direction that from $head(x_j)$ to $tail(x_j)$, then it finds the new beginning of support searching scope. On the other hand, if we modify AC3.1 and run it in the other direction that from $tail(x_j)$ to $head(x_j)$, it will find a new end of support searching range. We use $end(x_i, a, c_{ij})$ to record the new end of support searching scope. Similar to $beginning(x_i, a, c_{ij})$, all the values after $end(x_i, a, c_{ij})$ will not be a support for (x_i, a) in c_{ij}.

The new algorithm, named AC3$_{be}$, runs AC3.1 in both directions at the preprocessing step and generates both *beginning* and *end* data structures. When seeking for a support, it searches from $beginning(x_i, a, c_{ij})$ to $end(x_i, a, c_{ij})$. The new algorithm narrows the support searching scope at both sides and maintains no additional data structure during search. The only incremental part is that it runs additional AC3.1 at preprocessing step. It is worth running that when solving some hard instances which need a number of branches, because AC3$_{be}$ benefits from the data structure *end* at each search tree node. Besides, the multi-directional residue supports can be easily integrated into AC3$_{be}$, because the data structure *residue* is independent to *beginning* and *end*. AC3$_{be}$ is designed for being maintained during search, so we eliminate the preprocessing part which can be easily generated from AC3.1 and present the seekSupport procedure of AC3$_{be}$ in

Algorithm 5. It is obvious that both $beginning(x_i, a, c_{ij})$ and $end(x_i, a, c_{ij})$ are residue supports of (x_i, a), so we check if they are still in $dom(x_j)$ before searching for a new support at lines 3 to 6. Narrowing the support searching scope is implemented at lines 7 and 8.

Algorithm 5 seekSupport$_{be}(x_i, a, c_{ij})$

```
1:  if residue(x_i, a, c_ij) ∈ dom(x_j) then
2:      return residue(x_i, a, c_ij);
3:  if beginning(x_i, a, c_ij) ∈ dom(x_j) then
4:      return beginning(x_i, a, c_ij);
5:  if end(x_i, a, c_ij) ∈ dom(x_j) then
6:      return end(x_i, a, c_ij);
7:  b ← next(beginning(x_i, a, c_ij), dom(x_j));
8:  while b is not after end(x_i, a, c_ij) do
9:      if (a, b) satisfies c_ij then
10:         residue(x_i, a, c_ij) ← b;
11:         residue(x_j, b, c_ij) ← a;
12:         return b;
13:     else
14:         b ← next(b, dom(x_j));
15: return b;
```

Proposition 1: The worst case time complexity of AC3$_{be}$ is $O(ed^3)$ with space complexity $O(ed)$.

Proof: Although AC3$_{be}$ narrows the support searching scope, it still checks d values to find a support in the worst case, so the seekSupport$_{be}$ procedure needs $O(d)$ time. From Algorithm 2, we can see that seekSupport$_{be}$ is called at most d times in each revise procedure. From Algorithm 1, we can see that each arc (x_i, c_{ij}) will be revised at most d times, because at most d-1 values is removed from $dom(x_j)$ and each removal leads to a revision of arc (x_i, c_{ij}). We have $2e$ arcs in total, so the worst case time complexity of AC3$_{be}$ is $O(ed^3)$.

The space complexity of AC3$_{be}$ is bounded by the data structures *residue*, *beginning* and *end*. In each arc (x_i, c_{ij}), we have one residue support for each of the d values in $dom(x_i)$, so the data structure *residue* costs $2ed$ space. The other two data structures have the same space cost as *residue*. Therefore, the space complexity of AC3$_{be}$ is $O(ed)$. □

We summarize the differences among the data structures mentioned in the paper.

- $last(x_i, a, c_{ij})$: It stores the latest support for (x_i, a) in c_{ij} and needs to be maintained during search. When it is removed from $dom(x_j)$, AC3.1 searches for a new support from *last* to $tail(x_j)$.
- $residue(x_i, a, c_{ij})$: It stores the residue support for (x_i, a) in c_{ij} and does not need to be maintained during search. When it is removed from $dom(x_j)$, AC3$_{rm}$ searches for a new support from $head(x_j)$ to $tail(x_j)$.
- $beginning(x_i, a, c_{ij})$: It stores the first value that is a support for (x_i, a). Every value before *beginning* is not a support for (x_i, a). When *residue*, *beginning* and *end* are all removed from $dom(x_j)$, AC3$_{be}$ searches for a new support from *beginning* to *end*.

TABLE 1. Comparison of AC3 algorithm.

Algorithm	*last*	*residue*	*m-d*	*m-r*	*b-e*	time	space
AC3						$O(ed^3)$	$O(e)$
AC3.1	√					$O(ed^2)$	$O(ed)$
AC3.2	√	√	√			$O(ed^2)$	$O(ed)$
AC3$_r$		√				$O(ed^3)$	$O(ed)$
AC3$_{rm}$		√	√			$O(ed^3)$	$O(ed)$
AC3$_{rmk}$		√	√	√		$O(ed^3)$	$O(ked)$
AC3$_{be}$		√	√		√	$O(ed^3)$	$O(ed)$

- $end(x_i, a, c_{ij})$: It stores the last value that is a support for (x_i, a). Every value after *end* is not a support for (x_i, a).

V. RELATED WORKS AND DISCUSSION

The most related algorithms, AC3.1 and AC3$_{rm}$ have been introduced in section 3. We discuss other coarse-grained AC algorithms in this section.

- The original AC3 algorithm does not need any of the data structures mentioned in this paper. If we remove lines 1, 2, 6 and 7 of Algorithm 4, we have the seeking support procedure of AC3.
- The AC3$_r$ [24] algorithm is the first one that uses residue supports in original AC3 algorithm, which are not maintained during search. It is a simple and effective improvement of AC3. AC3$_{rm}$ makes use of multi-directionality in AC3$_r$. If we remove line 7 of Algorithm 4, we have the seeking support procedure of AC3$_r$.
- Based on AC3$_{rm}$, AC3$_{rmk}$ [25] stores k residue supports for each (x_i, a, c_{ij}). Before searching for a new support, it checks whether any one of these k residues is still in $dom(x_j)$. If none of them is, it searches for a new one and add it to the residue support list. The list is implemented by a FIFO queue with size k. When adding a new residue into the list and the list is full, the earliest added residue will be removed. AC3$_{rm2}$ has been shown to save both CPU time and constraint checks on some CSP instances.
- Exploring multi-directional residues in AC3.1, the AC3.2 [16] algorithm is considered as a combination of AC3.1 and AC3$_{rm}$. The comparison [17] between AC3.1, AC3.2 and AC3$_{rm}$ shows that AC3.2 is more efficient than AC3.1, but both them are outperformed by AC3$_{rm}$ due to the heavy data structure being maintained.

Table 1 summarizes the differences between these coarse-grained AC algorithms. *last* is whether the algorithm uses data structure *last* and maintains it during search. *residue* is whether the algorithm uses the residue support technique. *m-d* is whether the algorithm uses the multi-directionality technique. *m-r* is whether the algorithm uses the multiple residue support technique. *b-e* is the new technique proposed in this work, which searches for a support between *beginning* and *end*.

VI. EXPERIMENTS

The experiments were run on a PC with Intel(R) Core(TM) i5-3210M CPU @2.5GHz, 4GB RAM, JDK 1.7. We have compared AC3$_{be}$ with AC3$_{rm}$ and AC3$_{rm2}$ on some benchmark problems. The performance of maintaining these AC algorithms for finding the first solution or proving unsatisfability is measured by CPU time (cpu) in seconds and the number of constraint checks (cc). The variable ordering heuristic is *dom/wdeg* [26] and value ordering is lexicographical. In the following tables, the integers in the brackets under instance names are the number of tested instances in that series. The best of each row is in bold. The $AC3_{rm2}$ algorithm is implemented with a static FIFO policy [25] and the later added residues are checked earlier. The numbers of constraint checks are present by kilo (K), million (M) and billion (B). Timeout (out) is set to 1200 seconds. We eliminated the instances where all algorithms are timeout.

TABLE 2. Results on random instances.

instance		$AC3_{rm}$	$AC3_{rm2}$	$AC3_{be}$
<40,8,753,0.1> (100)	cpu	**3.25**	3.87	3.64
	cc	32M	16M	**15M**
<40,11,414,0.2> (100)	cpu	**4.27**	5.15	4.84
	cc	49M	27M	**27M**
<40,16,250,0.35> (100)	cpu	**3.61**	4.38	4.14
	cc	57M	35M	**33M**
<40,25,180,0.5> (100)	cpu	**5.08**	5.65	5.53
	cc	102M	67M	**64M**
<40,40,135,0.65> (100)	cpu	**3.54**	3.98	3.87
	cc	112M	79M	**71M**
<40,80,103,0.8> (100)	cpu	**3.01**	3.44	3.36
	cc	166M	124M	**108M**
<40,180,84,0.9> (100)	cpu	**5.15**	5.58	5.43
	cc	412M	314M	**270M**

Table 2 lists the results of some random instances situated at the phase transition [29] of search with different domain sizes, different constraint densities and different constraint tightness. For each class $<n, d, e, t>$, n is the variables number, d is the domain size, e is the constraints number and t is the constraint tightness. The constraints of these random instances are defined in extension, so we generate the binary constraints by matrices where constraint checks are cheap. The results show that $AC3_{be}$ needs less constraint checks than the others. $AC3_{rm}$ is the best one on these instances, although its needs more constraint checks than the others. This is because the implementations of $AC3_{rm2}$ and $AC3_{be}$ are more complicate than that of $AC3_{rm}$, so the saving from constraint checks can not overcome the loss from implementation.

Table 3 lists the results on Radio Link Frequency Assignment Problem (RLFAP), Job-Shop problem and Queens-Knights problem, which are benchmark instances from the competition of constraint solvers.[1]

- The RLFAP is the task of assigning frequencies to a number of radio links. More details can be found in [27].
- The Job-Shop problem is the task assigning jobs to resources at particular times. More details can be found in [28].
- The Queens-Knights problem is the task of putting on a chessboard of size $n \times n$, q queens and k knights such that no

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

TABLE 3. Results on real-world, patterned and academic problems.

instance		$AC3_{rm}$	$AC3_{rm2}$	$AC3_{be}$
RLFAP scens11 (10)	cpu1	27.1	29.4	**26.3**
	cpu2	**21.4**	26.8	23.8
	cc	307M	215M	**160M**
RLFAP scens11-f4	cpu1	40.1	42.9	**38.5**
	cpu2	**31.0**	38.5	34.6
	cc	486M	334M	**243M**
RLFAP scens11-f6	cpu1	2.97	3.13	**2.79**
	cpu2	**2.28**	2.77	2.53
	cc	36M	26M	18M
Job-Shop (41)	cpu1	84.56	83.75	**57.99**
	cpu2	**33.55**	38.86	36.12
	cc	2.4B	1.9B	**0.5B**
Job-Shop e0ddr1-10-by-5-1	cpu1	13.8	14.7	**8.4**
	cpu2	**6.36**	7.78	6.78
	cc	400M	371M	54M
Job-Shop e0ddr2-10-by-5-10	cpu1	1069	991	**705**
	cpu2	**402**	454	444
	cc	33B	23B	**6B**
Queens-Knights (14)	cpu1	13.24	13.15	**4.11**
	cpu2	1.82	2.15	**1.70**
	cc	122M	120M	**35M**
Queens-Knights 50-5-add	cpu1	69.0	68.3	**17.4**
	cpu2	5.79	6.40	**4.60**
	cc	649M	645M	**162M**
Queens-Knights 50-5-mul	cpu1	102	101	**34**
	cpu2	18.27	22.13	**17.97**
	cc	942M	912M	**285M**

two queens can attack each other and all knights form a cycle. More details can be found in [26].

We present the average results and some representatives results in Table 3. The constraints of these instances are originally defined in intension. The cpu time is sensitive to the cost of a constraint check, so we also convert the constraints into matrices (offline). cpu1 is the time cost of the instances with original form of constraints and cpu2 is the time cost of the instances with matrix constraints. From the results, we can see that $AC3_{be}$ saves a number of constraint checks over the existing algorithms. When the constraints are in original form, $AC3_{be}$ is the most efficient one in cpu time. When the constraint checks are cheap, $AC3_{rm}$ is the best one on RLFAP and Job-Shop problems. But on the Queens-Knight instances, $AC3_{be}$ is the best even though the cost of a constraint check is cheap.

We further investigated some representative instance to explain why $AC3_{be}$ saves this number of constraint checks. We found that some of the constraints in these instances are highly structured. For instance, on the first constraint between x_0 and x_1 of *e0ddr1-10-by-5-1*, the *beginning* supports for the values in $dom(x_0)$ are $0, 1, 2, \ldots, 106$ respectively and the *end* supports are all 106. The *end* supports for the values in $dom(x_1)$ are $0, 1, 2, \ldots, 106$ respectively and the *beginning* supports are all 0. Obviously, searching from the *beginning* supports to the *end* supports on this constraint will save a number of constraint checks than searching from 0 to 106.

In general, $AC3_{be}$ is not suggested to be used when the constraint checks are cheap. Although its time complexity is same to that of the existing algorithms, it saves a number of constraint checks in practice, so it saves some cpu time when the constraint checks are relatively expensive. It saves up to 5 times constraint checks and up to 4 times cpu time.

VII. CONCLUSION

In this paper, we propose a new arc consistency algorithm, $AC3_{be}$, which is designed for being maintained during search. $MAC3_{be}$ narrows the support searching range to save constraint checks without maintaining additional data structure. Although the worst case time complexity and space complexity of $AC3_{be}$ is same to that of $AC3_{rm}$, the experimental results show that while solving some benchmark instances, $AC3_{be}$ saves a number of constraint checks in practice, so it also saves some cpu time. When the constraint checks are relatively expensive, it saves up to 4 times cpu time. It works well on some instances with structured constraints.

Arc consistency is the foundation of other local consistencies, such as maxRPC, RPC, singleton consistencies [31] and max restricted pairwise consistency (maxRPWC) [30]. This work shows $AC3_{be}$ avoids a number of redundant constraint checks, so potentially, it may bring some improvements for these consistencies where the check for a support is much more expensive than AC.

REFERENCES

[1] A. K. Mackworth, "Consistency in networks of relations," *Artif. Intell.*, vol. 8, no. 1, pp. 99–118, 1977.
[2] T. Balafoutis, A. Paparrizou, K. Stergiou, and T. Walsh, "New algorithms for max restricted path consistency," *Constraints*, vol. 16, no. 4, pp. 372–406, 2011.
[3] J. Guo, Z. Li, L. Zhang, and X. Geng, "MaxRPC algorithms based on bitwise operations," in *Proc. 17th Int. Conf. Principles Pract. Constraint Programm.*, 2011, pp. 373–384.
[4] J. Vion and R. Debruyne, "Light algorithms for maintaining max-RPC during search," in *Proc. 8th Symp. Abstraction, Reformulation, Approximation*, 2009, pp. 167–174.
[5] K. Stergiou, "Restricted path consistency revisited," in *Proc. 21st Int. Conf. Principles Pract. Constraint Programm.*, 2015, pp. 419–428.
[6] L. Bordeaux and E. Monfroy, "Beyond NP: Arc-consistency for quantified constraints," in *Proc. 8th Int. Conf. Principles Pract. Constraint Programm.*, 2002, pp. 371–386.
[7] P. Nightingale, "Consistency for quantified constraint satisfaction problems," in *Proc. 11th Int. Conf. Principles Pract. Constraint Programm.*, 2005, pp. 792–796.
[8] L. Bordeaux, M. Cadoli, and T. Mancini, "Generalizing consistency and other constraint properties to quantified constraints," *ACM Trans. Comput. Logic*, vol. 10, no. 3, p. 17, 2009.
[9] K. Stergiou, "Preprocessing quantified constraint satisfaction problems with value reordering and directional arc and path consistency," *Int. J. Artif. Intell. Tools*, vol. 17, no. 2, pp. 321–337, 2008.
[10] C. Likitvivatanavong, Y. Zhang, S. Shannon, J. Bowen, and E. C. Freuder, "Arc consistency during search," in *Proc. 20th Int. Joint Conf. Artif. Intell.*, 2007, pp. 137–142.
[11] D. Sabin and E. C. Freuder, "Contradicting conventional wisdom in constraint satisfaction," in *Proc. 11th Eur. Conf. Artif. Intell.*, 1994, pp. 125–129.
[12] R. Mohr and T. C. Henderson, "Arc and path consistency revisited," *Artif. Intell.*, vol. 28, no. 2, pp. 225–233, 1986.
[13] C. Bessière, "Arc-consistency and arc-consistency again," *Artif. Intell.*, vol. 65, no. 1, pp. 179–190, 1994.
[14] C. Bessiére, E. C. Freuder, and J.-C. Régin, "Using constraint metaknowledge to reduce arc consistency computation," *Artif. Intell.*, vol. 107, no. 1, pp. 125–148, 1999.
[15] C. Bessière and J. C. Régin, R. H. C. Yap, and Y. Zhang, "An optimal coarse-grained arc consistency algorithm," *Artif. Intell.*, vol. 165, no. 2, pp. 165–185, 2005.
[16] C. Lecoutre, F. Boussemart, and F. Hemery, "Exploiting multidirectionality in coarse-grained arc consistency algorithms," in *Proc. 19th Int. Conf. Principles Pract. Constraint Programm.*, 2003, pp. 480–494.
[17] C. Lecoutre and F. Hemery, "A study of residual supports in arc consistency," in *Proc. 20th Int. Joint Conf. Artif. Intell.*, 2007, pp. 125–130.
[18] M. R. C. van Dongen, "Saving support-checks does not always save time," *Artif. Intell. Rev.*, vol. 21, no. 3, pp. 317–334, 2004.
[19] J. R. Ullmann, "Partition search for non-binary constraint satisfaction," *Inf. Sci.*, vol. 177, no. 18, pp. 3639–3678, 2007.
[20] H. Li, Y. Liang, J. Guo, and Z. Li, "Making simple tabular reduction works on negative table constraints," in *Proc. 27th AAAI Conf. Artif. Intell.*, 2013, pp. 1629–1630.
[21] R. Wang, W. Xia, R. Yap, and Z. Li, "Optimizing simple tabular reduction with a bitwise representation," in *Proc. 25th Int. Joint Conf. Artif. Intell.*, 2016, pp. 787–793.
[22] J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J.-C. Régin, and P. Schaus, "Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets," in *Proc. 22nd Int. Conf. Principles Pract. Constraint Programm.*, 2016, pp. 207–223.
[23] H. Verhaeghe, C. Lecoutre, and P. Schaus, "Extending compact-table to negative and short tables," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 1–7.
[24] C. Likitvivatanavong, Y. Zhang, J. Bowen, and E. C. Freuder, "Arc consistency in MAC: A new perspective," in *Proc. CPAI Workshop Held With CP*, 2004, pp. 93–107.
[25] C. Lecoutre, C. Likitvivatanavong, S. Shannon, R. Yap, and Y. Zhang, "Maintaining arc consistency with multiple residues," *Constraint Programm. Lett.*, vol. 2, pp. 3–19, Jan. 2008.
[26] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, "Boosting systematic search by weighting constraints," in *Proc. 16th Eur. Conf. Artif. Intell.*, 2004, pp. 146–150.
[27] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners, "Radio link frequency assignment," *Constraints*, vol. 4, no. 1, pp. 79–89, Feb. 1999.
[28] N. Sadeh and M. S. Fox, "Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem," *Artif. Intell.*, vol. 86, no. 1, pp. 1–41, 1996.
[29] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre, "Random constraint satisfaction: Easy generation of hard (satisfiable) instances," *Artif. Intell.*, vol. 171, nos. 8–9, pp. 514–534, 2007.
[30] C. Bessière, K. Stergiou, and T. Walsh, "Domain filtering consistencies for non-binary constraints," *Artif. Intell.*, vol. 172, nos. 6–7, pp. 800–822, 2008.
[31] C. Bessiére, S. Cardon, R. Debruyne, and C. Lecoutre, "Efficient algorithms for singleton arc consistency," *Constraints*, vol. 16, no. 1, pp. 25–53, 2011.

HONGBO LI received the Ph.D. degree in computer science from Jilin University, China. He is currently a Post-Doctoral Researcher with the School of Computer Science and Information Technology, Northeast Normal University, Changchun, China. His research interests include constraint programming, local consistencies and heuristics for solving constraint satisfaction problems.