Information Processing Letters 70 (1999) 175–184

Information Processing Letters

www.elsevier.com/locate/ipl

Arc consistency revisited

Yangjun Chen [1]

IPSI Institute, GMD GmbH, 64293 Darmstadt, Germany

Received 11 November 1998; received in revised form 3 May 1999
Communicated by L. Boasson

Abstract

In this paper, we propose a new arc consistency algorithm, AC-8, which requires less computation time and space than AC-6 and AC-7 proposed by Bessière et al. (1994, 1995). The main idea of the optimization is the divide-and-conquer strategy, thereby decomposing an arc consistency problem into a series of smaller ones and trying to solve them in sequence. In this way, not only the space complexity but also the time complexity can be reduced. The reason for this is that due to the ahead of time performed inconsistency propagation (in the sense that some of them are executed before the entire inconsistency checking has been finished), each constraint subnetwork will be searched with a gradually shrunk domain. In addition, the technique of AC-6 (Bessière, 1994) can be integrated into our algorithm, leading to a further decrease in computational complexity. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Algorithms; Constraint satisfaction problem; Constraint network; Arc consistency; Probabilistic analysis

1. Introduction

Many problems in artificial intelligence can be seen as special cases of a general NP-complete problem [3] that has been called the "consistent-labeling problem" by Haralick et al. [4–6], the "satisfying assignment problem" by Gaschnig [7] and the "constraint satisfaction problem" by Fikes and others [8,9].

A constraint satisfaction problem can be defined as follows [10].

- $N = \{i, j, \ldots\}$ *is the set of nodes, with* $|N| = n$,
- $D = \{b, c, \ldots\}$ *is the set of labels,*
- $E = \{(i, j) \mid (i, j)$ *is an edge in* $N \times N\}$, *with* $|E| = e$,
- $D_i = \{b \mid b \in D$ *and* (i, b) *is admissible*$\}$, *with* $|D_i| = a_i$,
- R_1 *is a unary relation, and* (i, b) *is admissible if* $R_i(b)$,
- R_2 *is a binary relation, and* (i, b)–(j, c) *is admissible if* $R_{ij}(b, c)$.

The constraint satisfaction problem is to find one or more n-tuples in $D \times D \times \cdots \times D$ which satisfy the given relations. For example, in the graph coloring problem, there are only binary constraints and $R(n_1, n_2)$ is the set of all pairs of colors (a, b) such that $a \neq b$, for all pairs of adjacent nodes n_1 and n_2.

Since the problem is NP-complete, it has been suggested that a preprocessing step be applied to eliminate local (node, arc and path) inconsistencies before any attempt is made to construct solution. These ideas are significant because such inconsistencies would otherwise have been repeatedly discovered by any backtrack search. Especially, such techniques have found wide application in constraint logic program-

[1] Email: yangjun@darmstadt.gmd.de.

ming [11], pattern recognition, image analysis and artificial intelligence [12–14].

By node consistency, only the unary relations on the different nodes are checked and the values satisfying these unary constraints are kept in the domain of each node. The arc consistency algorithm consists of checking the consistency of labels for each couple of nodes linked by a binary constraint and removing the labels that cannot satisfy this local condition [1,15, 16]. Path consistency algorithms ensure that any pair of labeling (i, b)–(j, c) allowed by a direct relation is also allowed by all paths from i to j [17–20]. In this paper, we discuss only the arc consistency problem.

The idea of the arc consistency algorithm introduced by Mohr and Henderson [19] is based on the notion of support. (The notion of support was first defined by Mackworth [10].) We say that a label b at node i has a support from node j (j not equal to i) if there exists a label c at j such that $(b, c) \in R_{ij}$. As long as label b at node i has a minimum of support from the labels at each of the other nodes, b is considered a viable label for node i. But once there exists a node at which no remaining label satisfies the required relation with b, then b can be eliminated as a possible label for node i. To make the support evident, each arc–label pair is assigned a counter (denoted counter$[(i, j), b]$ for the arc from i to j with label b at node i) to indicate to what extend a label at some node is supported by another node. As a matter of fact, the first step of algorithm AC-4 given by Mohr and Henderson is devoted to the computation of such counters, constructing a set (denoted S_{jc} for label c at node j) for each label to store those labels that are supported by it and a global set (denoted *LIST*) to store initial inconsistent labels. In the second step, the inconsistency is propagated and eliminated iteratively based on the data structures constructed in the first step. In this way, the arc consistency can be finally obtained. The time and space complexities of AC-4 are both $O(ea^2)$ (on the assumption that $a_i = a$ for each D_i). Recently, some new results have been achieved by Bessière [1] and Bessière et al. [2]. In [1], a space-optimal algorithm (named AC-6) is proposed, which finds supports for a label dynamically and requires only $O(ea)$ space. In [2], a refined version of AC-6 (named AC-7) is developed based on a general inference schema, which utilizes the bidirectional property of binary constraints to remove redundant checks. (By bidirectionality, we mean that if a label b at some node i supports a label c at another node j, then c also supports b.) In fact, AC-7 improves the time complexity of AC-6 by a constant factor (see Section 4.2).

In our method, the bidirectional property is also utilized. But we use it in a different manner. That is, we use it by assigning the data structure in the initialization phase. The bidirectionality allows us to improve the time complexity by a constant factor (see AC-7 [2] for comparison). Another observation is that whenever a label is eliminated, all those labels supported only by it can be immediately removed and should not be considered any more, which make the decomposition of a constraint network possible. More exactly, we can partition R_2 into a collection of subsets in some way and regard each of them (with the corresponding node domains) as a subproblem. Then, we apply AC-6 to each successively and every time take only a more shrunk domain into account. Based on a probabilistic analysis, we derive that both the average time and space complexities can be reduced to $O(na)$ using our algorithm.

In the next section, we present the refined arc consistency algorithm. In Section 3, we prove the correctness of this algorithm. In Section 4, we analyze the computational complexities of the algorithms. Section 5 is a short conclusion.

2. Algorithm description of AC-8

An arc consistency problem can be simply denoted by $AC(n, D, R_2)$, where $D = \{D_1, D_2, \ldots, D_n\}$ with each D_i being the finite set of possible labels for node i and R_2 represents the set of binary constraints considered. A binary constraint (or relation) R_{ij} between node i and j is a subset of the Cartesian product $D_i \times D_j$ that specifies the allowed (compatible) pairs of labels for i and j. Beginning with Montanari [1, 19], a binary constraint R_{ij} is usually represented as a Boolean matrix with $|D_i|$ rows and $|D_j|$ columns by imposing an order on the node domains. Value **true** at row b, column c, denoted $R_{ij}(b, c)$, means that the pair consisting of the bth label of D_i and cth label of D_j is compatible; value **false** means that the pair is not allowed. In this paper, we consider only those networks with $R_{ij}(b, c) = R_{ji}(c, b)$.

For our purpose, we partition R_2 into $R_2^1, R_2^2, \ldots, R_2^{n-1}$ with the following property:

R_2^1 contains all constraints of the form R_{1i} in R_2;
R_2^2 contains all those constraints of the form R_{2j}, where R_{2j} does not appear in R_2^1;
$\vdots$
R_2^{n-1} contains all those constraints of the form $R_{(n-1)k}$, where $R_{(n-1)k}$ does not appear in $R_2^1 \cup \cdots \cup R_2^l \cup \cdots \cup R_2^{n-2}$.

Then, we define the following subproblems:

$AC(n, D^1, R_2^1)$,
$AC(n, D^2, R_2^2)$,
$\vdots$
$AC(n, D^{n-1}, R_2^{n-1})$,

where $D^1 = D = \{D_1^1, \ldots, D_n^1\}$ (i.e., $D_i^1 = D_i$, $i = 1, \ldots, n$), D^2 represents the shrunk domain after $AC(n, D^1, R_2^1)$ is solved (i.e., $D^2 = \{D_1^2, \ldots, D_n^2\}$), $\ldots$, and $D^{n-1} = \{D_1^{n-1}, \ldots, D_n^{n-1}\}$ represent the shrunk domain after $AC(n, D^1, R_2^1), \ldots,$ and $AC(n, D^{n-2}, R_2^{n-1})$ have been solved. Obviously, what we want is to solve them, using AC-6, successively with the following features:
(1) $D = D^1 \geqslant D^2 \geqslant \cdots \geqslant D^{n-1} \geqslant D^n$,
(2) D^n is arc consistent.
Here, D^n represents the shrunk domain after all $AC(n, D^l, R_2^l)$ $(l = 1, 2, \ldots, n-1)$ have been solved. However, if we simply apply AC-6 to each, the accumulating results may be incorrect due to the fact that the inconsistency propagation for a subproblem may not be done completely only according to the current constraint subnetwork (i.e., some R_2^l, $1 \leqslant l \leqslant n-1$). Therefore, for each inconsistency propagation, more constraints should be considered. To this end, we define two procedures:

initialization(n, D^l, R_2^l) and

propagation$(n, D^l, R_2^1 \cup \cdots \cup R_2^l)$.

These yield solutions to $AC(n, D^l, R_2^l)$, where initialization(n, D^l, R_2^l) is a modified version of the initialization part of AC-6, while propagation$(n, D^l, R_2^1 \cup \cdots \cup R_2^l)$ is just a copy of the propagation part of AC-6 but with an implicit difference. More exactly, by initialization(n, D^l, R_2^l), the symmetry will be utilized to speed up the computation; and by propagation$(n, D^l, R_2^1 \cup \cdots \cup R_2^l)$, not only R_2^l, but also $R_2^1, \ldots, R_2^{l-1}$ will be considered for the inconsistency propagation. That is, the inconsistency propagation will not be restricted to the current R_2^l. Instead, all those subnetworks (with the shrunk domains) arranged prior to R_2^l will be re-checked. In the following, we will give this algorithm (named "Modified-AC-6"). It will be embedded in AC-8 to provide solutions to each subproblem. As we will see later, elaborating in this way, we can improve the computational efficiency by an order of magnitude (see Section 4.1 for a probabilistic analysis).

Principally, the following "Modified-AC-6" works in a similar way to AC-6 to check a subnetwork: $AC(n, D^l, R_2^l)$ $(l = 1, 2, \ldots, n-1)$. (Note that $D^l = \{D_1^l, \ldots, D_n^l\}$.) That is, it is also a two-phase algorithm: an initialization process and a propagation process. Only the bidirectional property of the binary constraint is employed to avoid many redundant checks. As with AC-6, the following data structures are utilized:

- A table M^l of Booleans keeps track of which labels of the initial domain are in the current domain or not ($M^l(i, b) =$ **true** means $b \in D_i^l$).
- $S_{jc} = \{(i, b) \mid (j, c)$ is the first encountered label in D_j^l supporting (i, b) on $R_{ij}\}$.
- A *LIST* contains labels deleted from the domain but for which the inconsistency propagation has not been performed.

In addition, we use a new data structure $T^l(i, b)$ to support the usage of the bidirectional property of binary constraints, which guarantees that (i, b) is inserted at most once in some S_{jc} for node j.

The procedure remove(c, D_j^l) eliminates label c from D_j^l, while "nextsupport" is used to find a next support from a node for some label whenever its current support from this node is removed due to inconsistency.

```
procedure Modified-AC-6
 1 LIST := Empty;
 2 for each $(i, b) \in D^l$ do $M^l(i, b) :=$ true;
       $T^l(i, b) :=$ true; $S_{ib} :=$ Empty_set;
   {initialization$(n, D^l, R_2^l)$}
 3 for each $R_{ij} \in R_2^l$ do {
```

```
4   **for** $(i,b)$ where $b \in D^l_i$ **do** {
5      **if** $T^l(i,b) = $ **true then**
       (* “$T^l(i,b) = $ **true**” indicates that $(i,b)$
          has not yet been inserted into any $S_{jc}$. *)
6         {
7          $c :=$ the first element of $D^l_j$;
           nextsupport($i, j, b, c, empty\text{-}mark$);
8          **if** $empty\text{-}mark = $ **true**
           (* “$empty\text{-}mark = $ **true**” indicates that
              $(i,b)$ has no supports from $j$. *)
9          **then** remove($b$, $D^l_i$); $M^l(i,b) :=$ **false**;
              Append($LIST$, $(i,b)$)
10         **else** {Append($S_{jc}$, $(i,b)$);
                 $T^l(i,b) :=$ **false**;
11               **if** $T^l(j,c) :=$ **true then**
                    Append($S_{ib}$, $(j,c)$);
                    $T^l(j,c) :=$ **false**;}
                    (* using symmetry *)
12        }}
13  **for** $(j,c)$ where $c \in D^l_j$ **do** {
14     **if** $T^l(j,c) = $ **true then**
15        {
16         $b :=$ the first element of $D^l_j$;
           nextsupport($j, i, c, b, empty\text{-}mark$);
17         **if** $empty\text{-}mark = $ **true**
18         **then** remove($c$, $D^l_j$); $M^l(j,c) :=$ **false**;
              Append($LIST$, $(j,c)$)
19         **else** {Append($S_{ib}$, $(j,c)$);
                 $T^l(j,c) :=$ **false**;
20               **if** $T^l(i,b) :=$ **true then**
                   Append($S_{jc}$, $(i,b)$);
                   $T^l(i,b) :=$ **false**;}
21        }}
22  **for** $(i,b)$ where $b \in D^l_i$ **do**
    (* After $R_{ij}$ has been checked, $T^l(i,b)$
       will be reset for each $(i,b)$ for the next call. *)
23     $T^l(i,b) :=$ **true**;
24  **for** $(j,c)$ where $c \in D^l_j$ **do**
    (* After $R_{ij}$ has been checked, $T^l(j,c)$
       will be reset for each $(j,c)$. *)
25     $T^l(j,c) :=$ **true**;}

  {propagation($n$, $D^{l'}$, $R^1_2 \cup \cdots \cup R^l_2$)}
26 **while** $LIST$ is not empty **do**
27   {
28    choose $(j,c)$ from $LIST$ and
      remove $(j,c)$ from $LIST$;
29     **for** $(i,b) \in S_{jc}$ **do**
30       {
31        remove $(i,b)$ from $S_{jc}$;
32        **if** $(j,c) \in S_{ib}$
33        **then** {remove $(j,c)$ from $S_{ib}$;
34               $d := c$;
                 nextsupport($i, j, b, d, empty\text{-}mark$);
35               **if** $empty\text{-}mark = $ **true**
36               **then** remove($b$, $D^l_i$);
                    $M(i,b) :=$ **false**;
                    Append($LIST$, $(i,b)$)
                 (* the propagation will not be
                    restricted to the current $R^l_2$. *)
37               **else** Append($S_{jd}$, $(i,b)$);
38               }
39       }
40   }
```

Principally, this algorithm works in a similar way as AC-6. That is, it is also a two-phase algorithm: an initialization process and a propagation process. In addition, the technique of dynamic supports used in AC-6 is adopted to save space. The first difference is in lines 10–11 and 19–20, where both S_{ib} and S_{jc} are increased if (i,b)–(j,c) are compatible, while in AC-6, only one of them, say S_{ib}, is increased. In this way, each linked node pair $\{i,j\}$ will be visited only once instead of two times (one for (i,j) and the other for (j,i)). The second difference is the usage of $T^l(i,b)$, which guarantees that (i,b) is inserted only once in some S_{jc} for node j (see lines 11 and 20).

During the initialization process, each S_{jc} contains at most one (i,b) for each linked node i, which is the first encountered label supported by (j,c). In the subsequent propagation process, such a label will be added to another S_{jd} if (j,c) appears in *LIST* and (j,d) is the first label supporting (i,b) after c and has not been removed. This principle applies to each label at each node and therefore, each label supported by (j,c) will be dynamically inserted into S_{jc} at most once. Thus, the size of S_{jc} used by AC-6 is much smaller than that used by AC-4. Furthermore, due to the decomposition strategy in our method, each time only one subnetwork is considered, the space complexity can be reduced to $O(na)$ in this way (see Section 4.1 for a probabilistic analysis).

Finally, the following procedure is used to find the first label in D^l_j after c (including c), which supports

(i, b) on R_{ij}. This procedure is wholly the same as that used by AC-6.

```
procedure nextsupport($i, j, b, c$, empty-mark)
(* inputs: $i, j, b, c$; outputs: $c$, empty-mark *)
  {
   if $c \in D_j^l$ then
     {
      empty-mark := false;
      while $M^l(j, c) =$ false do
         $c :=$ the label following $c$;
      while not $R_{ij}(b, c)$ and empty-mark = true do
         if $c$ is not the last label in $D_j^l$ then
            $c := next(c, D_j^l)$;
         else empty-mark := true;
     }
   else empty-mark := true;
  }
```

In the above procedure, $next(b, D_i^l)$ returns the first not-removed label after b in D_i^l if b is not the last label in D_i^l.

In the following, we give our AC-8 algorithm, in which both procedures initialization(n, D^l, R_2^l) and propagation$(n, D^l, R_2^1 \cup \cdots \cup R_2^l)$ are called iteratively and each time only a subnetwork with shrunk domains is considered. However, as mentioned earlier, the inconsistency propagation has to be done "globally". In this way, the processes for the consistency checking and the inconsistency propagation are interleaved with each other, leading to a drastic reduction both in time and space complexities (see Section 4.1).

```
procedure AC-8
  {
   LIST := Empty; $D^1 := D$;
   for each $(i, b) \in D^1$ do
      $M^1(i, b) :=$ true;
      $T^1(i, b) :=$ true; $S_{ib} :=$ Empty_set;
   decompose $R_2$ into $R_2^1, R_2^2, \ldots, R_2^{n-1}$;
   for $l = 1, \ldots, n-1$ do
     {
      initialization$(n, D^l, R_2^l)$;
      propagation$(n, D^l, R_2^1 \cup \cdots \cup R_2^l)$;
      (* For inconsistency propagation,
         $R_2^1 \cup \cdots \cup R_2^l$ should be considered. *)
      let the resulting domain be $D^{l+1}$;
      let the resulting tables $M$ and $T$ be $M^{l+1}$ and
         $T^{l+1}$, respectively;
     }
  }
```

From AC-8, we see that after each execution of initialization(n, D^l, R_2^l) and propagation$(n, D^l, R_2^1 \cup \cdots \cup R_2^l)$, the corresponding domain will be reduced to D^{l+1}, which is arc consistent with respect to $R_2^1 \cup \cdots \cup R_2^l$. In addition, we do not reproduce M^{l+1} and T^{l+1}, which are only a renaming of M^l and T^l.

3. Correctness of AC-8

In this section, we give a complete proof of the correctness of AC-8. First, we show that AC-8 builds an arc consistent solution. Then we prove that the solution is complete. For exposition, we use D_i^{l+1} $(i = 1, \ldots, n)$ to denote the remaining node domains after the lth iteration of the main **for** loop and $D_i^1 = D_i$. Accordingly, D_i^n $(i = 1, \ldots, n)$ correspond to the arc consistent solution.

Proposition 3.1. *Let D_i^n $(i = 1, \ldots, n)$ be the solution found by* AC-8. *Then D_i^n $(i = 1, \ldots, n)$ are arc consistent.*

Proof. For ease of explanation, we assume that each constraint network corresponds to a complete graph. That is, each pair of nodes is linked by an edge. If two nodes are not connected, we add a **true** constraint between them. From algorithm AC-8, we can see that on the first iteration of the main **for** loop only the consistency of edges $(1, 2), (1, 3), \ldots, (1, n)$ is checked on R_2^1. Since the corresponding data structures for all nodes are established, the propagation of inconsistency can be performed immediately with D_1^1 as a pivot (see Fig. 1(a)). Then D_i^1 $(i = 1, \ldots, n)$, the remaining domains after the first iteration of the main **for** loop, will have the property that the labels in each D_k^1 $(k = 2, \ldots, n)$ is consistent with D_1^1.

On the second iteration of the main **for** loop, edges $(2, 3), (2, 4), \ldots, (2, n)$ will be checked on R_2^2 and the node domains are shrunk to D_i^2 $(i = 1, \ldots, n)$ with the property that each D_k^2 $(k = 3, \ldots, n)$ is consistent with D_1^2 and D_2^2, and at the same time D_1^2 and D_2^2 are consistent with each other (see Fig. 1(b)). Note that on this iteration, the inconsistency may be propagated to

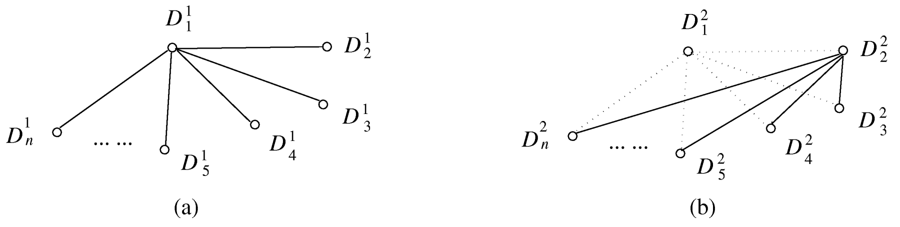

Fig. 1. Illustration for inconsistency propagation.

D_1^2 in terms of *support relationships* already collected in sets S_{jc} (on the preceding iteration). It is why on the second propagation we should consider $R_2^1 \cup R_2^2$ instead of R_2^2 alone. The same analysis applies to the other $n-3$ iterations of the main **for** loop. Hence, D_i^n $(i=1,\dots,n)$ have the property that D_n^n is consistent with all D_k^n $(k=1,2,\dots,n-1)$ and at the same time D_k^n $(k=1,2,\dots,n-1)$ are consistent with each other. In other words, D_k^n $(k=1,2,\dots,n)$ are arc consistent. □

The other question is: is any solution lost by propagating inconsistency earlier? The following proposition shows that the arc consistent solution found by AC-8 is complete.

Proposition 3.2. *Let* D_i^n $(i=1,\dots,n)$ *be the arc consistent solution found by* AC-8. *Then* D_i^n $(i=1,\dots,n)$ *are arc complete.*

Proof. The completeness follows from the fact that any eliminated label b at some node i has $M(i,b) \neq$ **true**. Such a label is removed either due to the lack of support from some other node (see lines 9 and 18 in Modified-AC-6); or due to the inconsistency propagation (see line 35 in Modified-AC-6); in this case, another (j,c') supporting it cannot be found. Therefore, such a label can not belong to any solution. Thus, the solution found by AC-8 is the largest arc consistent solution. □

4. Computational complexity

In this section, we analyze the average computational complexities. First, we compare AC-8 and AC-6 in Section 4.1. Then, in Section 4.2, we show that AC-7 improves AC-6 only by a constant factor.

4.1. *Probabilistic analysis of* AC-8 *and* AC-6

The constraint satisfaction problems can be represented by their *relations matrix* $[T_{km}^{ij}]$ (undefined for $i=j$), a bit-matrix such that element $T_{km}^{ij}=1$ iff the kth value for node i is consistent with the lth value for node j. Otherwise bit $T_{km}^{ij}=0$. This is essentially a truth-table representation of all relations between pairs of nodes. To simplify the description of the results of the analysis, we shall assume that D_i has the same size a and $\mathrm{Prob}(T_{km}^{ij}=1)=p$ for all i, j, k and m. That is, the probability of compatibility of any two labels for any two nodes equals p. Note that this assumption will not affect the correctness of our analysis results due to the following consideration. Let $\mathrm{Prob}(T_{km}^{ij}=1)=p_{km}^{ij}$. For any network problem with $0 \leqslant p_{km}^{ij} < 1$, we can always find another p $(0 \leqslant p < 1)$ such that $p_{km}^{ij} \leqslant p$ for all i, j, k and m.

First, we analyze the average time complexity of AC-6, which has not been reported elsewhere. Let $q=1-p$ be the probability of incompatibility of any two labels for any two nodes. Then, the expected number of checks performed on an iteration of the inner **for** loop of AC-6's initialization part is

$$\begin{aligned} &p+2qp+3q^2p+\cdots+kq^{k-1}p+\cdots \\ &\quad +aq^{a-1}p+aq^a \\ &\quad =\frac{p}{1-q}\left[\frac{1-q^a}{1-q}-aq^a\right]+aq^a \leqslant \frac{1}{p}. \end{aligned} \tag{1}$$

Therefore, its average time complexity can be given by

$$2\sum_e a\frac{1}{p}=\frac{2}{p}ea. \tag{2}$$

(Note that each edge (i,j) will be checked two times, one for $i \to j$ and the other for $j \to i$.)

Now we consider the average time complexity of AC-6's propagation part. To this end, we partition the propagation process into a series of phases. The first phase is defined as the process from the beginning to the moment when all the unallowed labels found in the initialization part are removed from *LIST* (known as *Waiting-List* in [1]). The second phase is from the end of the first phase to the moment when those elements inserted into *LIST* during the first phase are removed. In the same way, we can define the lth phase ($l > 2$). Then, we estimate the expected number of elements removed during each phase.

Consider an edge (i, j). The probability that each label (i, b) is supported by node j is

$$P = pq^{a-1} + p^2q^{a-2} + \cdots + p^kq^{a-k} + \cdots + p^aq^0.$$

Then, after the execution of the propagation part of AC-6, the size of each arc consistent node domain becomes P^na. Therefore, in total, for each node, there are $(1 - P^n)a$ labels removed from *LIST*. Let $Q = 1 - P$ be the probability that a label (i, b) is not supported by node j. Then we have

$$\begin{aligned} &n(1 - P^n)a \\ &\quad = Qna + QPna + QP^2na + \cdots + QP^{a-1}na. \end{aligned}$$

Note that the kth item: $QP^{k-1}na$ $(1 \leqslant k \leqslant n)$ in the polynomial corresponds justly to the expected number of elements removed during the kth phase. This is because the elements removed during the kth phase are those put in *LIST* during the $(k - 1)$th phase. An element e is put in *LIST* if and only if it is supported by some element e' of node i and at the same time e' becomes inviable and e has no more other supports from i. The probability that e is supported by e' during the $(k - 1)$th phase is P^{k-1} and the probability of e' becoming inviable is Q. Therefore, the probability that an element is inserted into *LIST* is QP^{k-1} during the $(k - 1)$th phase. Thus, $QP^{k-1}na$ is the expected number of elements removed during the kth phase.

Further, since arc–label pairs $[(i, j), b]$ have at most one support (j, c) with (i, b) belonging to S_{jc}, each S_{jc} contains on average one label (with probability P) for each linked node. Thus, for each removed label (i, b), there will be on average d_iP labels checked, where d_i represents the vertex degree at node i. Accordingly, the expected number of labels to be checked for any removed label (from *LIST*) is

$$\frac{1}{n}(d_1P + d_2P + \cdots + d_nP).$$

Hence, the average time complexity of the kth phase is on the order

$$\begin{aligned} &\frac{1}{n}\bigl(d_1P + d_2P + \cdots + d_nP\bigr) \cdot QP^{k-1}na \\ &\quad = \sum_{i=1}^{n} d_iPQP^{k-1}a, \end{aligned} \tag{3}$$

and the total average time complexity of its propagation part is

$$\sum_{k=1}^{n}\sum_{i=1}^{n} d_iQP^ka = \sum_{i=1}^{n} eQP^ka = \mathrm{O}(ea). \tag{4}$$

The average space complexity of AC-6 is not much better than its worst-case complexity and can be estimated as follows. Since during the initialization process of AC-6 each label will be inserted into one S_{jc} structure once (with probability P) for each of its adjacent nodes, the size of S_{jc} sets should be

$$\sum_{i=1}^{n} ad_iP = Pea = \mathrm{O}(ea). \tag{5}$$

This implies that its average space complexity cannot be below this quantity.

In the following, we analyze the computational complexity of our algorithm and show that both the average time and space complexities are $\mathrm{O}(na)$. First, we consider the computational complexity of all initialization(n, D^l, R_2^l)'s. As discussed above, the time complexity of initialization(n, D^1, R_2^1) should be on the order

$$\sum_{d_1} a\frac{1}{p} = \frac{1}{p}ad_1, \tag{6}$$

where d_1 represents the vertex degree at node 1, since in this process only the edges in R_2^1 are checked. Similarly, the average time complexity of initialization (n, D^l, R_2^l) can be given as

$$\frac{1}{p}P^{l-1}ad_l, \tag{7}$$

where d_l represents the vertex degree at node l.

Therefore, the total average time complexity of all initialization(n, D^l, R_2^l)'s is

$$\sum_{l=1}^{n-1} \frac{1}{p} P^{l-1} a d_l = \mathrm{O}(na). \tag{8}$$

In order to analyze the time complexity of propagation$(n, D^l, R_2^1 \cup \cdots \cup R_2^l)$, we partition the propagation process as for AC-6.

Consider propagation(n, D^1, R_2^1). After its execution, only the size of node 1 becomes $P^n a$, while the other nodes i $(i = 2, \ldots, n)$ are all shrunk to Pa. Therefore, in total, $(1 - P^n)a + (n-1)(1-P)a$ labels will be eliminated from *LIST* during this process. (In the following discussion, $(1 - P^n)a + (n-1)(1-P)a$ is denoted $N(1)$.) By a simple computation, we have

$$\begin{aligned}&(1 - P^n)a + (n-1)(1-P)a \\ &\quad = nQa + QPa + QP^2a + \cdots + QP^{n-1}a. \end{aligned}\tag{9}$$

This implies that the expected number of labels removed in the first phase is nQa and for the kth phase $(k = 2, \ldots, n)$, the expected number is $QP^{k-1}a$. Define $d_i(l)$ to be a function, representing the number of edges incident with i and visited by propagation$(n, D^l, R_2^1 \cup \cdots \cup R_2^l)$. Then, the expected number of labels to be checked for a removed label (from *LIST*) is

$$\frac{1}{n}\big(d_1(1)P + d_2(1)P + \cdots + d_n(1)P\big).$$

Accordingly, the average time complexity of propagation(n, D^1, R_2^1) can be estimated as

$$\begin{aligned}&\frac{1}{n}\left(\sum_{i=1}^{n} d_i(1)\right) nQa + \sum_{k=2}^{n} \frac{1}{n}\left(\sum_{i=1}^{n} d_i(1)\right) QP^k a \\ &\quad = \sum_{i=1}^{n} d_i(1)QPa + \sum_{k=1}^{n-1}\sum_{i=1}^{n} d_i(1)\frac{QP^{k+1}a}{n}. \end{aligned}\tag{10}$$

Further, after propagation$(n, D^2, R_2^1 \cup R_2^2)$ has been performed, both the sizes of node 1 and 2 become $P^n a$. The sizes of the other nodes i $(i = 3, \ldots, n)$ are all $P^2 a$. Therefore, in this process there are $N(2)$ labels removed from *LIST*, where

$$\begin{aligned}N(2) &= 2(1 - P^n)a + (n-2)(1-P^2)a - N(1) \\ &= (n-1)QPa + QP^2a + \cdots + QP^{n-1}a. \end{aligned}\tag{11}$$

From this, we know that during this process, the expected number of labels removed in the first phase is $(n-1)QPa$ and the expected number for the kth phase $(k = 2, \ldots, n)$ is $QP^k a$. In addition, we notice that in initialization$(n, D^2, R_2^1 \cup R_2^2)$ node 1 has not been checked. Therefore, labels of the $(1, j)$ will not appear in the first phase but may appear in the subsequent phases. Then, the average time complexity of propagation$(n, D^2, R_2^1 \cup R_2^2)$ is

$$\sum_{i=2}^{n} d_i(2)QP^2a + \sum_{k=2}^{n-1}\sum_{i=1}^{n} d_i(2)\frac{QP^{k+1}a}{n}. \tag{12}$$

In general, we have the following sum as the average time complexity of propagation$(n, D^l, R_2^1 \cup \cdots \cup R_2^l)$:

$$\sum_{i=l}^{n} d_i(l)QP^l a + \sum_{k=l}^{n-1}\sum_{i=1}^{n} d_i(l)\frac{QP^{k+1}a}{n}. \tag{13}$$

Therefore, the total average time complexity of all propagation$(n, D^l, R_2^1 \cup \cdots \cup R_2^l)$'s $(l = 1, 2, \ldots, n)$ can be given by

$$\sum_{l=1}^{n}\sum_{i=l}^{n} d_i(l)QP^l a + \sum_{l=1}^{n}\sum_{k=l}^{n-1}\sum_{i=1}^{n} d_i(l)\frac{QP^{k+1}a}{n}. \tag{14}$$

Note that

$$\sum_{i=l}^{n} d_i(l) \leqslant \sum_{i=1}^{n} d_i(l) \leqslant 2ln \quad \text{for } 1 \leqslant l \leqslant n.$$

Therefore, sum (14) is less than (15):

$$\begin{aligned}&\sum_{l=1}^{n} 2nlQP^l a + \sum_{l=1}^{n}\sum_{k=l}^{n-1} 2lQP^{k+1}a \\ &\quad \leqslant 2Qna\sum_{l-1}^{n} lP^l + 2Qpa\sum_{l=1}^{n}\sum_{k=l}^{n-1} lP^k. \end{aligned}\tag{15}$$

Finally, we give the following inequality to complete the time complexity analysis.

$$\begin{aligned}&\sum_{l=1}^{n}\sum_{k=l}^{n-1} lP^k \\ &\quad \leqslant \int_1^n \int_x^{n-1} xP^y \,\mathrm{d}y\,\mathrm{d}x \\ &\quad = \frac{1}{\ln P}\int_1^n \big[xP^y\big]_x^{n-1}\,\mathrm{d}x \end{aligned}$$

$$= \frac{1}{\ln P}\left(\frac{n^2 P^{n-1}}{2} - \frac{1}{2}P^{n-1}\right) - \frac{1}{(\ln P)^2}\left(nP^n - \frac{1}{\ln P}P^n - P + \frac{1}{\ln P}P\right) = \mathrm{O}(n).$$

From the above analysis, we know that the average time complexity of all propagation$(n, D^l, R_2^1 \cup \cdots \cup R_2^l)$'s is also $\mathrm{O}(na)$.

The average space complexity of AC-8 can be computed by investigating S_{jc}'s structure and its incrementation. On the first iteration of the main **for** loop of AC-8, the total size of S_{jc} sets is at most the number of arc–label pairs involved. Therefore, the corresponding space complexity is

$$SC_1 = d_1 a + (d_1 - 1)a \quad \text{(* see Fig. 1(a) *)}. \tag{16}$$

On the second iteration of the main **for** loop, the total size of S_{jc} sets becomes

$$SC_2 = d_1 Pa + d_2 Pa + (d_2 - 2)Pa \quad \text{(* see Fig. 1(b) *)}. \tag{17}$$

In general, on the lth iteration of the main **for** loop, the total size of S_{jc} sets will be changed to

$$SC_l = d_1 P^{2l+1}a + d_2 P^{2l+1}a + \cdots + d_l P^{2l+1}a + (d_l - l)P^{2l+1}a. \tag{18}$$

Therefore, the entire space complexity of AC-8 is on the order

$$SC = \max\{SC_1, SC_2, \ldots, SC_{n-1}\} = \mathrm{O}(na). \tag{19}$$

4.2. *Time complexity of AC-7*

In order to analyze the time complexity of AC-7 and to show why it improves AC-6 by a constant factor, we consider a simple network consisting of only one edge (i, j). For each arc consistency algorithm, the checks will be done for both $i \to j$ and $j \to i$. Without loss of generality, we assume that the checks for $i \to j$ precede those for $j \to i$ in AC-7. Then, no optimization can be done for $i \to j$ using the bidirectional property. For $j \to i$, AC-7 refines AC-6 in the following way. When a label for j, say (j, c) (supported by some label for i) is picked from "*SeekSupportStream*" (see [2]), another label will be sought due to the fact that the current support is removed. However, instead of invoking function "*SeekNextSupport*" (corresponding "nextsupport" in this paper) immediately, an inference will be made by calling function "*SeekInferableSupport*", in which S_{jc} will be searched to see whether it contains another label for i. If such a label, say (i, b), exists and at the same time it is a non-removed element, (j, c) will be inserted into S_{ib}. Otherwise, "*SeekNextSupport*" will be executed to find the next "first" support for (j, c) as AC-6 does. In this way, many checks for $j \to i$ will be saved by replacing them with "inferences". Therefore, in terms of the probabilistic analysis deriving (1) and (2), the checks for $i \to j$ is of the time complexity $\frac{1}{p}a$, while the number of checks for $j \to i$ is smaller than $\frac{1}{p}a$. This explains why AC-7 is better than AC-6.

5. Conclusions

In this paper, we have proposed a new algorithm, AC-8, to achieve arc consistency in a binary network. The key idea is the divide-and-conquer strategy and the decomposition of a constraint network into a series of smaller ones. On the one hand, for each subproblem, the incompatible checks are done with a gradually shrunk domain. On the other hand, the number of inconsistency propagations can be reduced dramatically. In this way, an optimal computational complexity can be obtained. In addition, the correctness of AC-8 is proved formally and the computational complexity comparison between AC-6 and ours is made based on a probabilistic analysis.

References

[1] C. Bessière, Arc-consistency and arc-consistency again, Artificial Intelligence 65 (1994) 179–190.

[2] C. Bessière, E.C. Freuder, J.-C. Regin, Using inference to reduce arc consistency computation, in: Proc. of 14th Internat. Joint Conf. on Artificial Intelligence (IJCAI-95), Montreal, Quebec, August 1995, pp. 592–598.

[3] R.M. Haralick, L.S. Davis, A. Rosenfeld, Reduction operations for constraint satisfaction, Information Sci. 14 (1978) 199–219.

[4] R.M. Haralick, L.G. Shapiro, The consistent labeling problem: Part 1, IEEE Trans. Pattern Anal. Machine Intelligence 1 (2) (1979) 173–184.

[5] R.M. Haralick, G.L. Elliott, Increasing tree search efficiency for constraint satisfaction problems, Artificial Intelligence 14 (1980) 263–313.

[6] R.M. Haralick, L.G. Shapiro, The consistent labeling problem: Part II, IEEE Trans. Pattern Anal. Machine Intelligence 2 (3) (1980) 193–203.

[7] J. Gaschnig, Performance measurement and analysis of certain search algorithms, Ph.D. Thesis, Department Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1979.

[8] R.E. Fikes, REF-ARF: A system for solving problems stated as procedures, Artificial Intelligence 1 (1970) 27–120.

[9] B.A. Nudel, Consistent-labeling problems and their algorithms: expected-complexities and theory-based heuristics, Artificial Intelligence 21 (1983) 135–178.

[10] A.K. Mackworth, Consistency in networks of relations, Artificial Intelligence 8 (1977) 99–118.

[11] P. Van Hentenryck, Constraint Satisfaction in Logic Programming, Logic Programming Series, MIT Press, Cambridge, MA, 1989.

[12] A.K. Mackworth, Interpreting pictures of polyhedral scenes, Artificial Intelligence 4 (1973) 121–137.

[13] U. Montanari, Optimization methods in image processing, in: Proc. IFIP Congress, North-Holland, Amsterdam, 1974, pp. 727–732.

[14] A. Rosenfeld, A. Hummel, S.W. Zucker, Scene labeling by relaxation operations, Computer Science TR-379, University of Maryland, College Park, MD, 1975.

[15] P. van Hentenryck, Y. Deville, C.-M. Teng, A generic arc-consistency algorithm and its specialization, Artificial Intelligence 57 (1992) 291–321.

[16] D.L. Waltz, Generating semantic descriptions from drawings of scenes with shadows, MAC AI-TR-271, MIT, Cambridge, MA, 1972.

[17] E.C. Freuder, Synthesizing constraint expressions, Comm. ACM 29 (1978) 24–32.

[18] C.-C. Han, C.-H. Lee, Comments on Mohr and Henderson's path consistency algorithm, Artificial Intelligence 36 (1988) 125–130.

[19] R. Mohr, T.C. Henderson, Arc and path consistency revisited, Artificial Intelligence 28 (1986) 225–233.

[20] U. Montanari, Networks of constraints: Fundamental properties and applications to pictureprocessing, Inform. Sci. 7 (1974) 95–132.