Maintaining Arc Consistency using Adaptive Domain Ordering*

Chavalit Likitvivatanavong
Cork Constraint Computation Centre
University College Cork, Ireland

Yuanlin Zhang
Department of Computer Science
Texas Tech University, Texas, U.S.

James Bowen and **Eugene C. Freuder**
Cork Constraint Computation Centre, University College Cork, Ireland

1 Introduction

Solving a Constraint Satisfaction Problem (CSP) by Maintaining Arc Consistency (MAC) [Sabin and Freuder, 1994] has been one of the most widely-used methods. Since Arc Consistency (AC) is enforced at every node in a search tree, its efficiency is critical to the whole algorithm. We propose a new MAC algorithm based on AC-3.1 [Zhang and Yap, 2001] in which the AC component is capable of starting from where it left off in its previous execution with low overhead. It has the following properties: (1) $O(ed^2)$ worst-case time complexity in any node *and* any branch of the search tree; (2) $O(ed)$ space complexity; (3) the ability to avoid a type of redundant constraint check called a *negative repeat*; (4) no recomputation and maintenance of its internal data structures upon backtrack.

2 Definitions

In this paper we consider CSPs with binary constraints and use D_X, n, e, and d to denote the current domain of variable X, the number of variables, the number of constraints, and the maximum domain size respectively. We also assume a domain to be totally-ordered and adopt a random-access doubly linked-list implementation. The terms head and tail denote the boundary of a domain. A constraint check between value $a \in D_X$ and value $b \in D_Y$ is denoted by $C_{XY}(a, b)$.

A *propagation-oriented backtrack search* algorithm for CSPs is the standard depth-first backtrack search framework augmented by some process that handles all constraint propagation and the maintenance of the internal data structures involved. The *search tree* is defined by associating each node with a variable assignment of the algorithm. *Node complexity* of the algorithm is the time complexity cost of the constraint propagation performed at each node. *Path complexity* is the aggregate cost for any path in the search tree, summing the cost of every node in succession, starting from the root, to a leaf. During search some constraint checks may be repeated many times even though the constraint processing component is optimal in a single execution. We define the following type of redundant check called a *negative repeat*. Negative repeats can be troublesome for hard problems that require a large amount of backtracking.

Definition 1 (Negative Repeat) A constraint check $C_{XY}(a, b)$ performed at time t during search is called a *negative repeat* with respect to Y if and only if:
(1) $C_{XY}(a, b)$ = false, and
(2) $C_{XY}(a, b)$ has been performed at time s where $s < t$, and
(3) b has been continuously present in the time interval (s, t).

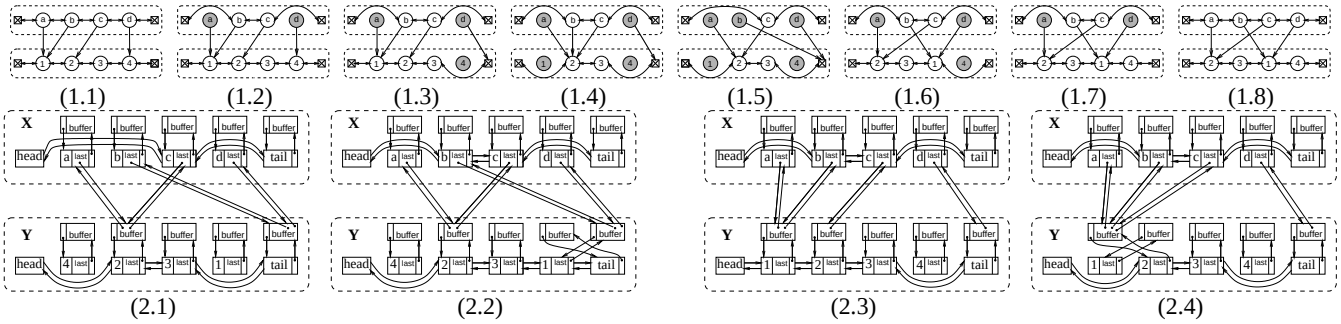
3 Adaptive Domain Ordering

In this section we introduce a new MAC algorithm called Adaptive Domain Ordering (ADO). The key feature is the dynamic rearrangement of variable domains after each backtrack. A pruned value is simply restored to the end of its domain, rather than its initial position. This technique makes the algorithm capable of avoiding *all* negative repeats as well as requiring no maintenance on its internal data structure. Unlike AC-3.1, on which the constraint processing unit of ADO is based, we associate the following two invariants for last(X,a,Y). First, the *safety invariant*: there exists no support of a in $\{c \in D_Y \mid c <$ last(X,a,Y)$\}$. Second, the *prospect invariant*: there exists a support of a in $\{c \in D_Y \mid c \geq$ last(X,a,Y)$\}$. ADO is a propagation-oriented backtrack search algorithm and behaves like MAC-3.1 for the most part. The main differences lie in the routines for removing a value (`remove`), finding a support, and restoring a value (`restore`). When a value is removed, any last that points to it will be moved to the next value. When no support for $a \in D_X$ is found in D_Y, last(X,a,Y) will be made to point to tail of Y. When a value is restored, all the last pointers that point to the tail will be made to point to the restored value instead. It can be proved that, given any path in a search tree, the worst-case aggregate complexity of `remove` is $O(ed^2)$. Due to space restriction we are not able to give detailed proofs of the complexity cost and correctness of ADO. An example trace of the algorithm is shown as follows.

Example Consider $D_X = \{a, b, c, d\}$, $D_Y = \{1, 2, 3, 4\}$ and C_{XY}=$\{(a, 1)$,$(b, 1)$,$(c, 2)$,$(d, 4)\}$ (allowed tuples). The result after the initial AC processing[1] is shown in Figure (1.1): last(X,a,Y)=1, last(X,b,Y)=1, last(X,c,Y)=2, and last(X,d,Y)=4. Suppose a and d are removed due to some external cause; their last values remain unchanged (1.2). Next, suppose 4 is removed. The result after AC processing is shown in (1.3). At

*This work has received support from Science Foundation Ireland under Grant 00/PI.1/C075.

[1] In truth, we only enforce directional AC on the arc (X,Y) to keep the example simple.



the next search level suppose 1 is removed. According to the algorithm, any last that points to 1 — including last(X,a,Y) — will be shifted to 2 (1.4). Figure (1.5) shows the result after the problem is made arc consistent. Now consider the network after backtrack. Since 1 is removed after 4, it must be restored before. Figure (1.6) shows the result after 1 is restored. All the last pointers that pointed to tail are moved to 1. Notice that b is restored as well because it was pruned at the same level as 1. Figure (1.7) shows the result after 4 is restored. Figure (1.8) shows the network after the search backtracks to the point where we started this example. Note how the last pointers and the domain ordering differ from (1.1). □

To make ADO efficient, we need another data structure associated with each value to account for the last pointers that point to it. This is called buffer. Its elements can be preallocated since a value has only one last pointer in any constraint. We then make a last pointer refer to a value's buffer instead of the value itself — i.e. $(a, X) \in$ buffer(b,Y) iff last(X,a,Y) points to buffer(b,Y). This allows a set of last pointers to be switched all at once just by rearranging related buffers. For example, consider Figure (2.1), a more detailed view of (1.5). When 1 is restored, we want to move all the pointers from tail to 1. This can be done simply by swapping the two buffers (2.2). As a result `restore` takes $O(1)$ time.

We use a similar technique for `remove`. We compare the buffer of the value to be removed with that of the next value in the current domain (or tail if there is none) and swap both buffers if the first contains more elements. For an example, consider Figure (2.3), which is the detailed view of Figure (1.3). When value 1 is removed, its buffer size is compared with the buffer size of value 2. Since it has more elements, we move pointers from the buffer for value 2 into that for value 1 and swap both buffers, which results in (2.4).

Given any path in a search tree, the worst-case aggregate complexity of `remove` using buffer with the above techniques is $O(ed \lg d)$. We can further reduce the cost by representing data in a buffer as a rooted tree and having the root of the smaller tree point to that of the larger one. In fact, this is the union-by-rank operation for the disjoint-set union problem. `remove` then takes $O(1)$ time. However, locating the value of a last pointer no longer takes constant time. By using path compression, its worst-case aggregate complexity is $O(ed\ \alpha(ed^2))$ where α is the inverse Ackerman's function, which is almost constant [Tarjan, 1975].

ADO has $O(ed^2)$ for node and path complexity since the cost related to buffer is subsumed by the standard cost of establishing arc consistency. Correctness follows because: (1) the prospect invariant states that there is a support somewhere for each value, thereby enforcing arc consistency; (2) the safety invariant states that no support is skipped. Note that because the problem is made arc consistent before the search starts, we can always expect a value to have a support. If a value is subsequently removed, it is only because all of its supports are pruned; these supports are still in the original domain. The existence of a support in the original domain is central in the correctness proofs, and without the AC preprocessing before the search starts ADO would not be correct.

4 Conclusions

We have designed ADO to explore the theoretical limits of MAC and, as far as we know, it achieves the best results and outperforms all other existing MAC algorithms. Specifically, ADO has $O(ed^2)$ worst-case time complexity in any node *and* any branch of the search tree while using only $O(ed)$ space. This resolves the trade-off between the two traditional implementations of MAC-3.1. The first one records every change made to last so that after backtrack the algorithm can start from the exact same state. Although both node and path complexity for this approach is $O(ed^2)$, its space complexity is $O(ed \min(n, d))$. The second approach resets last in every search node to keep the space at $O(ed)$, but this comes at the expense of path complexity, which becomes $O(ned^2)$.

Régin [2004] also aims to create a maintenance-free MAC algorithm that has the best features from the two implementations of MAC-3.1. The algorithm in [Régin, 2004] is both node and path optimal while using $O(ed)$ space. However, the last structure needs to be recomputed and updated after each backtrack besides the normal restoration of pruned values. It cannot avoid negative repeats. By contrast, ADO requires no recomputation and no update of its internal structure and is able to avoid negative repeats.

References

[R´egin, 2004] J.-C. R´egin. Maintaining arc consistency algorithms during the search with an optimal time and space complexity. In *CP-04 Workshop on Constraint Propagation and Implementation*, 2004.

[Sabin and Freuder, 1994] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of ECAI-94*, pages 125–129, 1994.

[Tarjan, 1975] R. E. Tarjan. Effi ciency of a good but not linear set union algorithm. *Journal of ACM*, 22(2):215–225, 1975.

[Zhang and Yap, 2001] Y. Zhang and R. H. C. Yap. Making AC-3 an optimal algorithm. In *Proceedings of IJCAI-01*, pages 316–321, 2001.