

# Quick Shaving

Olivier Lhomme

ILOG, 1681, route des Dolines, 06560 Valbonne, FRANCE  
olhomme@ilog.fr

## Abstract

Arc-consistency plays such a key role in constraint programming for solving real life problems that it is almost the only algorithm used for reducing domains. There are a few specific problems for which a stronger form of propagation, often called shaving, is more efficient. Nevertheless, in many cases, shaving at each node of the search tree is not worth doing: arc-consistency filtering is much faster, and the additional domain reductions inferred by shaving do not pay off. In this paper, we propose a new kind of shaving called QuickShaving, which is guided by the search. As QuickShaving may infer some additional domain reductions compared with arc-consistency, it can improve the search for a solution by an exponential ratio. Moreover, the advantage of QuickShaving is that in practice, unlike a standard form of shaving, the additional domain reductions deduced by QuickShaving come at a very low overhead compared with arc-consistency.

## Introduction

Arc-consistency plays a key role in constraint programming for solving real life problems. The arc-consistency filtering algorithms have been continuously improved for twenty years (Mohr & Henderson 1986; Bessière 1994; Bessière & Régin 1997; Bessière & Régin 2001; Zhang & Yap 2001) and are almost the only kind of filtering algorithms used for reducing domains in practical applications. There still remain a few specific application domains, *e.g.*, scheduling, for which a stronger form of propagation, often called shaving, is more efficient. The principle of shaving is quite simple: a value is tentatively assigned to a variable, and an arc-consistency filtering algorithm is applied. If an inconsistency is found, then the value can be safely removed from the domain of the variable. Otherwise, the value is kept in the domain of the variable.

There are basically two kinds of using shaving:

- *The optimistic use* which assumes that inferences done by shaving, although costly, will pay off. All the values of all the variable domains can be tested for shaving. Furthermore, when a value is removed from the domain, previous values found as consistent can be retested, because

now they possibly can be detected as inconsistent. If this process is repeated until a fixed point is reached, the constraint network is said to be Singleton Arc Consistent (SAC) (Debruyne & Bessière 1997).

- *The pessimistic use* which tries only some values in order not to spend too much CPU time. This approach is often used in practice. For example, in scheduling, a common strategy is to try only the values that are the bounds of the domains of some selected variables. On numeric CSPs, 3B-consistency (Lhomme 1993), which often helps a lot for solving difficult problems, is a kind of shaving only on the bounds of the domains.

The optimistic approach has recently attracted the attention of several researchers (Bartak & Erben 2004; Bessière & Debruyne 2004). Their aim is to improve the SAC computations by making the algorithm incremental. Thus, if SAC is achieved at each node of a search tree, the SAC computations are amortized along a branch of the search.

Nevertheless, in many cases, shaving on each value at each node of the search tree is not worth doing: arc-consistency filtering is much faster, and the additional domain reductions inferred by shaving do not pay off. We are more on the pessimistic side, and think that good heuristics for choosing the values to be tried for shaving also deserve attention.

In this paper, we propose a new inference mechanism that needs only a small overhead in time. It can be seen as a kind of shaving, and we call it QuickShaving. The principle is to test only some values for shaving that are selected by analyzing the failures that occur during the search.

Experiments show huge improvements for solving problems where the additional inferences do help, and very small overhead when the additional inferences do not help.

The paper is organized as follows. First, we briefly give the motivations for shaving. Then we present a new method that is called QuickShaving. Some experimental results are then given, and related work is discussed.

## Motivations for Shaving

Why doing shaving during search? Its interest is typically when the choice of a value, say  $x = 1$ :

- leads to a problem that is arc-consistent,

- but there is one variable, say  $y$ , deeper in the search tree, which cannot be assigned without leading to a problem being arc-inconsistent.

When the variable  $y$  is assigned, inconsistency is detected, but all parts of the tree between  $x$  and  $y$  must be fully explored before undoing the culprit assignment  $x = 1$ . If shaving is applied on  $y$  just after the assignment  $x = 1$ , the inconsistency can be detected immediately.

Consider the following example. Let  $x_1, \dots, x_n$  be  $n$  variables of a constraint problem. Let us consider a subproblem SP on variables  $x_1, x_{10}, x_{11}, x_{12}$ , with associated domains  $d(x_1) = \{1, 2, 3, 4\}, d(x_{10}) = \{1, 2, 3\}, d(x_{11}) = \{1, 2, 3\}, d(x_{12}) = \{1, 2, 3\}$ , and with difference constraints between each pair of variables in SP:  $x_1 \neq x_{10}, x_1 \neq x_{11}, x_1 \neq x_{12}, x_{10} \neq x_{11}, x_{10} \neq x_{12}, x_{11} \neq x_{12}$ .

Consider a backtracking algorithm that performs arc-consistency at each node and that uses static ordering of variables from  $x_1$  to  $x_n$ , and a value ordering that takes first the smallest value in the current domain.

As soon as  $x_1$  is assigned with value 1, 2 or 3, the subproblem is a pigeon hole problem<sup>1</sup>: it has no solution but arc-consistency is not able to see the inconsistency.

The search thus continues by assigning  $x_2, x_3, \dots, x_9$ . When assigning  $x_{10}$ , arc-consistency detects inconsistency. The search now needs to explore the whole subtree between  $x_2$  and  $x_9$  before backtracking to  $x_1$ .

If shaving on  $x_{10}$  is applied after  $x_1$  is assigned with value 1, 2 or 3, inconsistency is detected immediately and thus the useless exploration of the whole subtree between  $x_2$  and  $x_9$  is avoided.

## Quick Shaving

The main drawback of shaving is CPU time consumption. It tries to shave a lot of values in a blind way. This requires many computations, which are often useless:

- It is clearly a waste of CPU time when testing a value for shaving leads to an arc-consistent problem: the value still remains in the domain of the variable but the CPU time is spent.
- Even when the shaving succeeds in removing a value, the computations can be useless. For example, it is not efficient to waste CPU time to reduce the domain of  $x$  if we branch on  $x$  just after. (Of course the choice of the variable to branch may depend on the size of the domain and thus we may argue that sometimes a better informed heuristic may save exponential time but this is a side effect, and we will try to separate in our study the inferences and the heuristics.)

The time spent in all those useless shaving computations is in general much more than the time spent in useful shaving

computations. We will see that we can avoid most of them. First of all, we need to define the term “shavable”:

**Definition 1** A value  $a$  of a variable  $x$  is said to be shavable when AC is able to detect inconsistency on the problem  $P' = P \cup \{x = a\}$ .

More generally, a decision or a constraint  $c$  is said to be shavable when AC is able to detect inconsistency on the problem  $P' = P \cup c$

A perfect shaving algorithm would test for shaving the shavable values only. Unfortunately, it is not possible to know in advance which values are shavable. The strategy we adopt for using shaving in the search is the following: at depth  $k$ , test for shaving only the values that would be shavable at depth  $k+1$ . We need to have an oracle that predicts at depth  $k$  what will happen at depth  $k+1$ . Backtracking can be used as such an oracle; its advantage is that it does not cost additional CPU time. All we need is to keep which values are shavable at depth  $k+1$ . Then, when a backtrack to depth  $k$  occurs, we can test for shaving those values that are shavable at depth  $k+1$ .

Thus, shaving is tried only at backtrack, and only on values that are shavable in the next deeper node of the search tree. But where does the first shavable value come from? The answer is very simple. It comes from branching, when assigning a value leads to a direct failure. For example, let us assume the search is at depth  $k$ . The search algorithm branches on  $x = 2$ , and immediately fails. Thus, we know that  $x = 2$  is a shavable value at depth  $k$ , without having to test  $x = 2$  for shaving. When a backtrack at depth  $k - 1$  will occur, the shavable values at depth  $k$ , like  $x = 2$ , will be tested for shaving.

## Description of the algorithm

Figure 1 shows an example of how to implement a search algorithm with the QuickShaving principle. Procedure QS-Search receives as input parameters the constraint problem  $P$  and a constraint to add to  $P$  which is a branching decision. Procedure QS-Search also has an output parameter, *shavableList* which is a list of values that are shavable after adding the decision  $ct$  to  $P$ . Procedure QS-Search returns true or false, to indicate that a solution has been found or that there is no solution to the problem  $P \cup ct$ .

The first call to QS-Search gives a constraint that is always true for parameter  $ct$ .

If the decision  $ct$  leads to a failure, then we know that this decision is shavable. So, *shavableList* is initialized to the set containing this decision.

If there are no more variables to assign, the search is finished and a solution is found. Otherwise, a branching decision is chosen, for example an assignment of a value to a variable, and a recursive call to QS-search is done.

If the recursive call succeeds, a solution has been found. Otherwise, we must try the negation of the decision. For example, if the decision is an assignment of a value  $a$  to a variable  $x$ , its negation is the constraint  $x \neq a$ . But, before this second recursive call, we can benefit from the list of shavable decisions that have been found by the first recursive call. The function *shave* tests all those shavable decisions

<sup>1</sup>A pigeon hole problem is a problem where one must find a hole for each pigeon, such that each pigeon has its own hole, but there are  $n$  pigeons and  $n - 1$  holes. Such pigeon hole problems led to the introduction of the *allDifferent* constraint that solves this kind of problem in polynomial time by using a matching algorithm. But that is not the point here; we take this example only for illustration purposes and consider that we do not have *allDifferent* constraint.

for shaving. It adds constraints to the constraint problem and cleans the list shavable1 by side effects (see Figure 2).

Note that the function shave may also detect inconsistency of the constraint network: in this case, we can backtrack without trying the other branch of the alternative. We only have to report shavable1 in shavableList.

When the function shave succeeds, the second branch of the alternative is tried by a recursive call to QS-Search, and either a solution is found, or the shavable decisions found in both branches of the alternatives are merged in shavableList.

We chose to present an algorithm in a very simple form for illustration. It can be optimized in different ways. For example, in the real implementation a decision that has just led to a failure is not retried by the function shave.

We now study the properties of QuickShaving.

### Properties of Quick Shaving

We want to qualify the overhead QuickShaving may add to the use of arc-consistency. The following properties show that this overhead is small in theory.

**Property 1** *Let  $d$  be a branching decision that fails at depth  $k$  and which is shavable at depth  $g \leq k$ , but not shavable at depth  $g - 1$ . QuickShaving will shave this decision  $k - g$  times and will fail to shave it once (at depth  $g - 1$ ).*

**proof:** The proof is immediate: consider the branch of the search that leads to this decision. On this branch, QuickShaving only tests it for shaving on backtrack, and only when it succeeds in shaving it. ☺

The above property is crucial: it means that QuickShaving can be wrong in its prediction of which values to shave only once per value on a branch of a search. This will explain why, at worst, QuickShaving only leads to a small overhead in time on our experiments.

**Property 2** *When QuickShaving does not infer additional reductions compared with arc-consistency, the search does at most  $F$  additional AC-filtering calls compared with the same search without QuickShaving, where  $F$  is the number of leaves of the search tree encountered so far.*

**proof:** QuickShaving does no additional domain reductions compared with arc-consistency filtering. Thus, all the decisions that fails at a depth  $k$  are not shavable at depth  $k - 1$ . By Property 1, we thus have that there is at most one shaving for such a decision. Those decisions are those that lead to a fail, and thus are leaves of the search tree. ☺

The property 2 leads to an even stronger practical result:

**Corollary 1** *In practice, when QuickShaving does not infer additional reductions compared with arc-consistency, the overhead is small compared with the same search without QuickShaving*

**proof:** When QuickShaving does not infer additional reductions compared with arc-consistency, it only more or less doubles the CPU time spent in the leaves of the search tree. In practice, the complexity of the arc-consistency filtering is increasing with the number of still unassigned variables and with the size of the domains. At the bottom of the tree, domains are smaller and unassigned variables are fewer than at

```

procedure QS-Search(in P, in ct, out shavableList)
  add ct to P
  achieve arc-consistency
  if failure
    shavableList = {ct}
    return false
  if some variable is not assigned
    choose a branching decision: dec
    ret1 = QS-Search(P,dec, shavableList)
    if ret1 = true
      return true
    else
      if shave(P, shavableList)
        ret2 = QS-Search(P, not(dec), shavableList)
        if ret2 = true
          return true
        else
          shavableList = shavableList ∪ shavableList2
          return false
      else
        shavableList = shavableList
        return false
  else
    return true // solution found

```

Figure 1: Example of use of QuickShaving in search

the top of the tree. Thus, at the top of the tree, the cost of the arc-consistency filtering is greater than at the end of the tree. ☺

We will see that this corollary is confirmed by our experiments.

Now, we address the advantages of QuickShaving. Of course, the advantage wrt AC-consistency is clear:

**Property 3** *The use of QuickShaving in a search algorithm may avoid the exploration of an exponential number of nodes compared with AC-filtering.*

**proof:** Consider a static ordering on the branching decisions. Let us consider the decision that is taken at depth  $k$ . Assume:

- this decision fails at depth  $k$ ,
- it is shavable at depth  $g$ ,
- its negation also fails at depth  $k$ ,
- its negation is also shavable at depth  $g$ ,
- no other decision leads to a failure.

QuickShaving saves the exploration of  $d^{k-g}$  nodes of the search tree. ☺

The Section on experiments will help to give an idea of the practical efficiency of QuickShaving.

### Experiments

In the introduction, we have seen an example that shows that QuickShaving may lead to reduced computation time by an exponential factor. As expected, our method leads to an arbitrarily large improvement in efficiency for such cases. But this is due to the specific structure of this kind of cases.

The experiments we performed have two objectives:

```

procedure shave(inout P, inout shavableList)
  foreach dec in shavableList
    if arc-consistency ( $P \cup \text{dec}$ ) fails
       $P = P \cup \text{not}(\text{dec})$ 
      if arc-consistency(P) fails
        return false
    else
      remove dec from shavableList
  return P

```

Figure 2: shave function

- We want to show that, even for problems with no structure at all, i.e. on random problems, QuickShaving may lead to drastic reductions of CPU time.
- We also want to show that, when the inferences of QuickShaving do not help the search, or when QuickShaving achieves only very few domain reductions the overhead is very small. That is to say that we want to confirm experimentally Property 2 and Corollary 1.

All the experiments have been made with ILOG Solver 6.1 (ILOG 2005)

### Random set of tuples

First we consider a test that is an adaptation of a local search benchmark on Boolean variables: the NK model (Kauffman 1993). The problems of the first test involve 30 variables. All the constraints are given in extension by their allowed set of tuples, and have the same arity (6). Each constraint is defined over randomly chosen variables. The Cartesian product size is  $2^6$ . We take randomly  $2^5$  tuples in each table. Thus, the density of each constraint is around 0.5. The advantage of this model is that, by varying only one parameter, the number of constraints, we vary the difficulty in solving the problem. The CPU time for finding the first solution is reported in the following table.

#ct	AC time	QS time
2	0.04	0.11
3	0.06	0.05
4	1.492	0.08
5	1.342	0.06
6	0.03	0.03
7	0.03	0.04

- #ct is the number of constraints.
- “AC” corresponds to arc-consistency.
- “QS” represents QuickShaving.
- Times are expressed in seconds.

We can see a phase transition for 4 or 5 constraints without QuickShaving.

We then increase the number of variables to 50:

#ct	AC time	QS time
2	> 600	0.23
3	> 600	0.16
4	> 600	0.13
5	> 600	0.14
6	0.08	0.04
7	0.04	0.04

This time, we cannot get results without QuickShaving in less than 10 minutes for problems with 2, 3, 4 or 5 constraints. We typically have an exponential behavior, and QuickShaving gives an exponential improvement in time.

### Testing the overhead

For some classic puzzles like N-queens problems or magic squares, which are very dense, only a little additional pruning is achieved by QuickShaving. Those problems allow us to quantify the overhead of QuickShaving. For example, for the N-queens problem, with minSizeDomain heuristic, we get the following results.

#queens	AC time	QS time
50	2.55	2.08
60	2.12	2.25
70	1.42	1.14
80	2.02	2.13
90	5.15	5.36
100	3.89	3.86

The CPU times are close. Some runs are better with QuickShaving. This may be due to the fact that the heuristic is a little bit better informed. So we test the same problem with a static order for the variable, but with a smaller number of queens.

#queens	AC time	QS time
10	0.07	0.08
11	0.04	0.05
12	0.11	0.12
13	0.04	0.05
14	0.6	0.74
15	0.4	0.55
16	3.5	3.9
17	2.1	2.5
18	14.9	17.1
19	1.12	1.24
20	78.0	89.0

This test is one of the worst cases we found for QuickShaving. We can see that the overhead compared with arc-consistency is small.

### Related Work

We already mentioned the different work done to make SAC incremental (Bartak & Erben 2004; Bessière & Debruyne 2004).

An original attempt to improve shaving has been proposed in (Lebbah & Lhomme 1998). It has been developed for continuous CSPs. While testing a value for shaving (this is not

exactly a value but a small interval near one of the bounds of the domain), an extrapolation method is periodically called. If the extrapolation method predicts that the network has a good chance to be consistent, and, thus, that the shaving will not be possible, the test for shaving is stopped. Unfortunately, extrapolation cannot be applied on discrete domains. Our method can thus be seen as a way to find another oracle other than extrapolation which works for discrete domains.

QuickShaving takes its roots in the different look-back techniques *Dependency Directed Backtracking* (Stallman & Sussman 1977), *Conflict-directed BackJumping* (Prosser 1995), *Dynamic Backtracking* (Ginsberg 1993), *Partial order Dynamic Backtracking* (Ginsberg & McAllester 1994), *Generalized Partial order Backtracking* (Bliek 1998), etc. Unlike forward techniques like arc-consistency filtering, look-back techniques infer domain reductions only when the values have been tried in the search. The method we propose in this paper adopts an approach similar to that of look-back techniques for choosing the value on which shaving is applied, that is, the use of the failures.

## Conclusion

In this paper, we have proposed QuickShaving, a new method for reducing domains.

QuickShaving can be seen as a shaving method; it is able to infer more domain reductions than arc-consistency and thus allows exponential time to be saved in a search for a solution. We observed such exponential gains in our experiments.

The specificity of QuickShaving is that it is able to predict which values have a good chance of being shavable. Only those values are tested for shaving. The good property of QuickShaving is that it can be wrong in its prediction only once per value on a branch of a search. This property is its main advantage compared with other shaving techniques like SAC: its overhead in time is quite small compared with arc-consistency filtering. Thus, it may be used as a default filtering algorithm.

## Acknowledgments

I would like to thank Christian Bliek, Michel Leconte, and Paul Shaw for a careful reading of this paper and Ulrich Junker and Philippe Refalo for fruitful discussions.

## References

- Bartak, R., and Erben, R. 2004. New algorithm for singleton arc consistency. In *FLAIRS-2004*. AAAI Press.
- Bessière, C., and Debruyne, R. 2004. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of CP'04 workshop on Constraint Propagation and Implementation*.
- Bessière, C., and Régin, J.-C. 1997. Arc consistency for general constraints networks: preliminary results. In *IJCAI'97*, 398–404.
- Bessière, C., and Régin, J.-C. 2001. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01*, 309–315.
- Bessière, C. 1994. Arc-consistency and arc-consistency again. *Artificial Intelligence* 65:179–190.
- Bliek, C. 1998. Generalizing partial order and dynamic backtracking. In *Proceedings of AAAI*.
- Debruyne, R., and Bessière, C. 1997. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 412–417.
- Ginsberg, M., and McAllester, D. A. 1994. Gsat and dynamic backtracking. In *International Conference on the Principles of Knowledge Representation (KR94)*, 226–237.
- Ginsberg, M. 1993. Dynamic backtracking. *Journal of Artificial Intelligence Research* 1:25–46.
- ILOG. 2005. *ILOG Solver 6.1 User's manual*. ILOG S.A.
- Kauffman, S. A. 1993. *The Origins of Order*. Oxford University Press.
- Lebbah, Y., and Lhomme, O. 1998. Acceleration methods for numeric csp. In *Proceedings of AAAI-1998*.
- Lhomme, O. 1993. Consistency techniques for numeric CSPs. In *IJCAI'93*, 232–238.
- Mohr, R., and Henderson, T. C. 1986. Arc and path consistency revisited. *Artificial Intelligence* 28:225–233.
- Prosser, P. 1995. MAC-CBJ: maintaining arc-consistency with conflict-directed backjumping. Research Report 95/177, Department of Computer Science – University of Strathclyde.
- Stallman, R. M., and Sussman, G. J. 1977. Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence* 9:135–196.
- Zhang, Y., and Yap, R. 2001. Making ac-3 an optimal algorithm. In *Proceedings of IJCAI'01*, 316–321.