Constraint-level advice for shaving

Radoslaw Szymanek, Christophe Lecoutre

Constraint-level Advice for Shaving

Radoslaw Szymanek[1] and Christophe Lecoutre[2]

[1] Artificial Intelligence Laboratory, EPFL, Switzerland.
radoslaw.szymanek@epfl.ch
[2] CRIL-CNRS UMR 8188, Universite d'Artois, Lens, France
lecoutre@cril.fr

Abstract. This work concentrates on improving the robustness of constraint solvers by increasing the propagation strength of constraint models in a declarative and automatic manner. Our objective is to efficiently identify and remove shavable values during search. A value is shavable if as soon as it is assigned to its associated variable an inconsistency can be detected, making it possible to refute it. We extend previous work on shaving by using different techniques to decide if a given value is an interesting candidate for the shaving process. More precisely, we exploit the semantics of (global) constraints to suggest values, and reuse both the successes and failures of shaving later in search to tune shaving further. We illustrate our approach with two important global constraints, namely alldifferent and sum, and present the results of an experimentation obtained for three problem classes. The experimental results are quite encouraging: we are able to significantly reduce the number of search nodes (even by more than two orders of magnitude), and improve the average execution time by one order of magnitude.

1 Introduction

Constraint Programming (CP) has become one of the dominant approaches to model and solve real-world combinatorial problems [1]. However, while CP has many success stories, it is believed that improving the usability of the technology is a key factor in its future success [2]. This is largely due to the fact that using constraints technology often requires considerable expertise in the use of its tools. In this paper, we aim at improving CP usability by using constraints semantics to augment propagation strength of the constraint model, without excessive computational cost, through means of shaving.

A shavable value is a value which, if assigned to the variable it is associated with followed by constraint propagation, entails an inconsistency. Shaving (introduced in the scheduling context [3, 4]) can be defined as the attempt of identifying and removing some shavable values. Shaving, as presented in [5], has two different flavors. The first one assumes that shaving always pays off no matter how many values are tested. Moreover, any shaved value will cause the re-execution of shaving for all variables which may then be susceptible to shaving again. Shaving applied in such a manner makes the problem Singleton Arc Consistent (SAC) [6].

In this work, we concentrate on a different flavor of shaving, which tries only some values in the shaving process. By reducing the number of values being evaluated, we hope to reduce the computational load without compromising too much on the quality of pruning. In the ideal case, we would like to achieve SAC by evaluating only values which are not SAC. The work presented in [5] points out the need for good heuristics which choose the values used in shaving. Here, we propose to use (global) constraints to suggest values to be used in the shaving procedure. In this way, we can utilize the constraint semantics to achieve a higher success ratio of shaving and an improved performance of search.

Our motivation is that any additional cost-effective propagation obtained without explicit user involvement will improve the quality of CP based approaches especially when the user is not fully aware of the intricate relationship between model and search. Although being different, this can be related (in terms of objectives) to recent works about constraint acquisition [7, 8]. Indeed, learning so-called implied constraints is just another means to improve the propagation capabilities of a constraint solver, without any expertise requirement on the users.

Interestingly, the work presented in [9] also proposes some form of constraint guided shaving. It is mainly implemented for global cardinality constraint (GCC) and as a result of successful shaving attempts, implied constraints in the shape of Among are added to the model. GCC is a generalization of alldifferent. Therefore, this approach is also concerned with alldifferent. In the context of alldifferent, this technique concentrates on values which, if successfully shaved, create Hall intervals which can lead to strengthening alldifferent propagation. This approach evaluates multiple values, therefore it is more costly in terms of finding values and shaving effort. Unfortunately, there is little experimental results to support the claim that shaving done in such a manner actually provides efficiency gains. In addition, in case of GCC while shaving, cheaper/weaker propagator of GCC is employed to reduce the cost of shaving as all values for cardinality variables are tried to be shaved.

We concentrate on proposing shaving in such a manner so tweaking the level of consistency strength is not required to recover costs, as well as we propose a complete framework where the effects of past multiple shaving attempts across multiple constraints are combined to improve future shaving attempts.

The remainder of this paper is organized as follows. After some preliminaries in Section 2, Section 3 presents the shaving framework. An illustration of how constraints can be used to propose values for shaving is presented in Section 4. Section 5 presents a detailed empirical evaluation. We conclude in Section 6.

2 Preliminaries

A Constraint Network (CN) P is a pair $(\mathscr{X}, \mathscr{C})$ where $\mathscr{X}$ is a finite set of n variables and $\mathscr{C}$ a finite set of e constraints. Each variable $X \in \mathscr{X}$ has an associated domain, denoted $dom(X)$, that contains the set of values allowed for X. Each constraint $C \in \mathscr{C}$ involves an ordered subset of variables of $\mathscr{X}$ and has

an associated relation[3], denoted $rel(C)$, which is the set of tuples allowed for this subset of variables. This subset of variables is the *scope* of C and is denoted $scp(C)$. The *arity* of a constraint is the number of variables in its scope. A *binary* constraint has arity 2.

A solution to a CN is an assignment of a value to each variable such that all the constraints are satisfied. A CN is said to be *satisfiable* iff it admits at least one solution. The Constraint Satisfaction Problem (CSP) is the NP-hard task of determining whether a given CN is satisfiable or not. A CSP instance is defined by a CN which is solved either by finding a solution or by proving unsatisfiability.

Usually, the domains of the variables of a given CN are reduced by removing inconsistent values, i.e. values that cannot occur in any solution. In particular, it is possible to filter domains by considering some properties of constraint networks. These properties are called domain-filtering consistencies [10, 11]. By exploiting consistencies, the problem can be simplified (and even, sometimes solved) while preserving solutions.

Given a consistency ϕ, a CN P is said to be ϕ-consistent iff the property ϕ holds on P. Enforcing a domain-filtering consistency ϕ on a CN means taking into account inconsistent values (removing them from domains) identified by ϕ in order to make the CN ϕ-consistent. The new obtained CN, denoted by $\phi(P)$, is called the ϕ-closure[4] of P. If there exists a variable with an empty domain in $\phi(P)$ then P is clearly unsatisfiable, denoted by $\phi(P) = \bot$.

A pair (X, a) with $X \in \mathscr{X}$ and $a \in dom(X)$ will be called a *value* (of P). The set of values of P that can be built from a constraint C is $values(C) = \{(X, a) \mid X \in scp(C) \wedge a \in dom(X)\}$. $P|_{X=a}$ denotes the CN obtained from P after removing all values but a from $dom(X)$. Shaving can then be defined as the attempt of identifying and removing some shavable values.

Definition 1. *Let P be a CN, and ϕ be a consistency. A value (X, a) of P is ϕ-shavable iff $\phi(P|_{X=a}) = \bot$.*

An attempt to shave value a from the domain of variable X is then performed in the following manner. First, variable X is assigned the value a. Second, the consistency ϕ is enforced. If in the process of reaching the consistency fix-point, one domain becomes empty, then it clearly indicates that assigning a to X does not lead to any solution. Therefore, it is possible to remove a from the domain of X. On the other hand, if assigning a to X does not entail a domain wipe-out, then the effects of constraint propagation (while enforcing ϕ) as well as the assignment of a to X must be retracted. The shaving attempt has failed. Sometimes, when the context is clear or unimportant, we will omit ϕ to simply refer to shavable values.

The most studied and employed consistency is generalized arc consistency (GAC), simply called arc consistency (AC) when constraints are binary. For a

[3] The introduction of $rel(C)$ does not prevent us from exploiting intensional representation of constraints.

[4] We assume here that $\phi(P)$ is unique. This is the case for usual consistencies [12].

formal definition, see e.g. [12]. Notice that a GAC-shavable value is a value that is not singleton arc consistent (SAC).

Definition 2. *Let P be a CN. A value (X, a) of P is singleton arc-consistent (SAC) iff $GAC(P|_{X=a}) \neq \bot$.*

Consider a CN composed of three variables X_1, X_2, X_3 such that $dom(X_1) = \{1, 3\}$, $dom(X_2) = \{1, 2\}$, and $dom(X_3) = \{2, 5\}$, and two constraints $C_1 : alldifferent(X_1, X_2, X_3)$ and $C_2 : X_3 = X_1 + X_2$. The first constraint imposes that all variables must be assigned different values, whereas the second one imposes that X_3 is equal to the sum of X_1 and X_2. If value $(X_1, 1)$ is tested for shaving then alldifferent and sum constraints will together discover inconsistency (when enforcing GAC) leading to the removal of 1 from $dom(X_1)$. This successful shaving attempt will cause further domain reductions making $dom(X_2) = \{2\}$ and $dom(X_3) = \{5\}$. On the other hand, if value $(X_1, 3)$ had been tested for shaving, no inconsistency would have been detected. In other words, $(X_1, 1)$ is shavable, whereas $(X_1, 3)$ is not.

Enforcing SAC on a given CN involves removing any value that is not singleton arc-consistent, i.e. any shavable value. This is a systematic approach which requires to consider each value in turn. Even if there exists some sophisticated approaches [13–15] to enforce SAC, this may be very time consuming. Maintaining such a consistency during search seems quite counter-productive. This is the reason why some limited forms of SAC have been devised such as bound SAC and existential SAC [16]. In this paper, contrary to previous works, we exploit the semantics of constraints to guide the shaving process.

3 Framework for Constraint-guided Shaving

In this section, we introduce the principles of constraint-guided shaving, before introducing a general algorithm and discussing some extensions.

3.1 Principles

Backtracking search is commonly employed for solving CSP instances. It corresponds to a depth-first search in order to instantiate variables and a backtracking mechanism when dead-ends occur.

With binary branching, at each step of the search, a pair (X, a) is selected where X is an unassigned variable and a a value in $dom(X)$, and two cases are considered: the assignment $X = a$ and the refutation $X \neq a$. Classically, we start by assigning variables before refuting values. We then explore a binary tree where left children correspond to variable assignments and right children to value refutations.

The motivations for the choice of binary branching (an alternative is d-way branching) are numerous. First, binary branching is commonly used in industry solvers. Second, there is a number of research work (e.g. [17]) which advocates the use of a binary branching scheme. Roughly speaking, binary branching is more

general as it does not prohibit switching to a different variable after exploring only one variable-value pair.

With shaving, at each node, we not only enforce a given consistency ϕ as usual to prune some portions of the search space[5], but also make some attempts to discover shavable values. This work incorporates a number of simple principles or techniques to increase the success ratio of shaving as well as the impact of shaved values on further pruning. The worst case scenario for shaving is trying many different values and not being able to shave them. This incurs only cost and does not give any benefit to the search. Besides, shaving in order to be efficient must at least shave some values from variable domains. However, please keep in mind that reducing the domains of the variables through shaving does not necessarily improve the overall search efficiency. The shaved values could have been removed with much smaller effort deeper in a search tree (even if it may be repeated several times).

In order to control the shaving overhead and to increase the effects of shaved values, we propose to exploit the semantics of constraints. More precisely, each constraint is asked to select one value to be used in shaving. Each constraint should aim at proposing one value which, if shaved, has the highest impact on the immediate pruning strength of the constraint. Moreover, we hope that increasing the pruning capabilities of the guiding constraints will in turn increase the propagation of the other constraints.

Another principle that we adopt, to limit shaving overhead, is the restriction of constraint advice for shaving only in search nodes which correspond to left children. The argument for this restriction is quite simple: the equality constraint (a variable assignment) added to reach the left child is usually much tighter than the negation of this constraint (a value refutation) added to reach the right child. Therefore, the difference between the root and the left child will most likely be larger than the difference between the root and the right one. We speculate that the left child has more chances to create new shaving capabilities than the child on the right.

Finally, the last technique to improve the success ratio of shaving is the use of a set (called $recentlyUnshaved$ in the algorithm below) which records all values for which shaving attempts have recently failed. If a constraint proposes the value (X, a) which belongs to this set then (X, a) is skipped, but it is also removed from the set. Therefore, if the value is proposed again later in the search then it may be tried again.

3.2 Algorithm

The pseudo-code for binary search with shaving is depicted in Algorithm 1. This is an algorithm with two embedded recursive calls. In this paper, we use the terms *search node*, *decision*, *wrong decision*, and *backtrack* as defined in [19]. This algorithm takes four parameters: P, $leftChild$, $recentlyShaved$, $recentlyUnshaved$

[5] For example, MAC [18] is the backtracking search algorithm that maintains (generalized) arc consistency at each step of search. So, we have $\phi = (G)AC$.

Algorithm 1: ϕ-ShavingSearch

Input:
P - the constraint network $(\mathscr{X}, \mathscr{C})$,
$leftChild$ - the Boolean specifying if P corresponds to a left child
Input/Output:
$recentlyShaved$ - the set of values that were recently shaved,
$recentlyUnshaved$ - the set of values that recently failed to be shaved
Output :
true/false to specify if a solution to P was found

```
1  P ← φ(P)
2  if P = ⊥ then
3      return false
4  if ∀X ∈ 𝒳, |dom(X)| = 1 then
5      return true
6  locallyShaved ← ∅
7  if leftChild = true then
8      foreach constraint C in 𝒞 do
9          (X, a) ← C.getValueForShavingAttempt()
10         if (X, a) ∈ recentlyUnshaved then
11             recentlyUnshaved ← recentlyUnshaved \ {(X, a)}
12         else
13             if φ(P|_{X=a}) = ⊥ then
14                 P ← φ(P|_{X≠a})
15                 if P = ⊥ then
16                     recentlyShaved ← recentlyShaved ∪ locallyShaved
17                     return false
18                 else
19                     locallyShaved ← locallyShaved ∪ {(X, a)}
20             else
21                 recentlyUnshaved ← recentlyUnshaved ∪ {(X, a)}
22 else
23     foreach (X, a) ∈ recentlyShaved do
24         if φ(P|_{X=a}) = ⊥ then
25             P ← φ(P|_{X≠a})
26             if P = ⊥ then
27                 return false
28         else
29             recentlyShaved ← recentlyShaved \ {(X, a)}
30             recentlyUnshaved ← recentlyUnshaved ∪ {(X, a)}
31 (X, a) ← selectVariableValue()
32 if φ-ShavingSearch(P|_{X=a}, true, locallyShaved, recentlyUnshaved) then
33     return true
34 if φ-ShavingSearch(P|_{X≠a}, false, locallyShaved, recentlyUnshaved) then
35     return true
36 recentlyShaved ← recentlyShaved ∪ locallyShaved
37 return false
```

that denote respectively the given constraint network, a Boolean value specifying if the current search node is a left child (i.e. reached after assigning a variable), the set of values which were recently successfully shaved, and the set of values which were recently attempted to be shaved without any success. The two first parameters are handled in the input mode whereas the parameters $recentlyShaved$ and $recentlyUnshaved$ are handled in the input/output mode. This algorithm returns true if there is a solution to the given constraint network.

Initially, the consistency ϕ is enforced (see line 1). If a failure is detected (see line 2), $false$ is returned since no solution can be found. Otherwise, if all domains of variables are singleton (see line 4), it means that a solution has been found. Notice that we suppose here that ϕ is a consistency that is at least as strong as backward checking, i.e. (at least) allows to detect any unsatisfied constraint involving variables which have all a singleton domain. This is quite a reasonable assumption.

If P corresponds to a constraint network reached on a left child (see lines 7 to 21), then we ask each constraint to propose one value for shaving attempt (line 9), and take it into account except if we recently failed to shave it (lines 10 and 11). If adding constraint $X = a$ and propagating (using consistency ϕ) makes the problem inconsistent, then we can shave value a from the domain of X and propagate this deletion. If this shaving makes the problem inconsistent (lines 14 and 15), then the $recentlyShaved$ set is updated and search is forced to backtrack. If shaving value (X, a) does not entail inconsistency then we can continue search and update the $locallyShaved$ set (line 19). Finally, if the shaving attempt was unsuccessful then the $recentlyUnshaved$ set is updated (line 21).

If P corresponds to a constraint network reached on a right child (see lines 22 to 30), we simply check for shaving all values in $recentlyShaved$. Indeed, if shaving guided by constraints is always performed after variable assignments, one can note that the results of these shaving attempts may influence shaving done in the remaining parts of the search (e.g. siblings). This is the reason why we use the $recentlyShaved$ set to perform shaving in right children. Each value (X, a) from this set is then attempted to be shaved. If (X, a) is shavable and removing it causes inconsistency then the search is forced to backtrack. On the other hand, if the shaving attempt was unsuccessful, then both $recentlyShaved$ and $recentlyUnshaved$ are updated (lines 29 and 30).

Here, we speculate that both children are similar enough to actually make it useful to use shavable values from left child when entering the child on the right. Moreover, if a value was successfully shaved in both children then this value is added to the set of shavable values of the root node (effectively, by not executing line 29 in the right child and executing line 36 upon exiting the parent of the right child). Therefore, the values which were successfully shaved deep in the left subtree will be tried in the right subtree. We restrict shaving speculation to only right children, as upon entering the left child the shavable list does not contain a value which could be used for speculation. The left child does not have a sibling which was executed earlier and all values which were successfully shaved by the parent node of the left child are still shaved.

To finish the description of the algorithm, we have to consider the recursive calls. Lines 31 to 35 allows to select a variable and a value for branching (using variable and value ordering heuristics) and to proceed to the left and right children. Before backtracking from the current node (line 37) the set *recentlyShaved* is updated (line 36) by adding to it all values which were successfully shaved in this search node.

3.3 Extensions

Guiding constraints Even if the algorithm is presented in such a way that any constraint participates to shaving, it may be more realistic to consider that only a subset of the constraints of the network are solicited for shaving attempts. For example, we may only consider (global) constraints whose semantics renders easy or natural such an exploitation. We discuss this aspect in the next section.

Quick Shaving It is rather easy to further incorporate the quick shaving technique proposed in [5]. In order to simplify the presentation of this incorporation to Algorithm 1, we use a global Boolean variable $leftChildWrongDecision$ which is set to true only when the left child was a wrong decision. To achieve this, we only need to insert the following instructions between lines 2 and 3:

if $leftChild = true$ **then**
  $LeftChildWrongDecision \leftarrow true$

When the left child led to a dead end immediately then the search proceeds to the following instructions (inserted between lines 33 and 34) which implement Quick shaving:

if *leftChildWrongDecision* **then**
  $locallyShaved \leftarrow locallyShaved \cup \{(X, a)\}$
  $recentlyUnshaved \leftarrow recentlyUnshaved \setminus \{(X, a)\}$
  $leftChildWrongDecision \leftarrow false$

In short, Quick shaving adds a value which led to a wrong decision to the *recentlyShaved* set just before the search exits the parent node of the wrong decision.

4 Constraint Guidance

We now discuss about constraint guidance for shaving. We concentrate on global constraints such as *sum* and *alldifferent* since they are commonly used in many problem classes.

We first need to introduce pruning events (see for example [20]) which can be specifically treated to speed up constraint propagation. The *bound* event occurs if the minimal or maximal value from the variable domain is removed. The *ground* event occurs if all but one value are removed from the domain. If the domain of a variable shrinks in another way then it qualifies as event *any*. It is often the case that the design of the constraint propagation algorithms makes it impossible to infer any additional domain pruning in case of occurrence of the event *any*. Therefore, we adapted our suggestion mechanisms within constraints to prefer/suggest values which cause a *bound* or *ground* event if a value is successfully shaved. This preference increases the chance of additional inferences based on just shaved values.

4.1 Alldifferent

Our example uses the *permutation* constraint to demonstrate how the internal data structures maintained by this global constraint can be used to propose values for shaving. A *permutation* constraint is applicable when the numbers of variables and values are equal and we wish to ensure that each variable takes a different value. Therefore, the *permutation* constraint can be regarded as a special case of the *alldifferent* constraint [21]. A filtering algorithm for *permutation* can be readily derived from the filtering algorithm for *alldifferent*. The *alldifferent* constraint maintains an internal data structure called the value graph to achieve generalized arc consistency (GAC) [21]. The value graph is a bipartite graph in which the edges link variables to values in their current domain. An example of the value graph is presented in Figure 1. This value graph can be efficiently reused to identify values that can be assigned to a small number of variables. In our example, thanks to the value graph an important observation can be made. We can observe that all variables can be assigned at least three different values, however value 1 can be assigned to only two variables. Based on this knowledge, we can choose variable X_1 and value 1 or variable X_2 and value 1 for shaving, hoping that the chances of successful shave will increase.

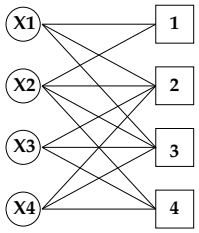

Fig. 1. An example of a value graph.

The suggestion mechanism of the *alldifferent* constraint considers all variables within a constraint scope which have a domain consisting of two elements,

as well as all values which are in the domain of only two variables. For any entity ε (either a variable or a value) such that $|dom(\varepsilon)| = 2$, it computes the following metric m, where $m = min(|dom(el_1)|, |dom(el_2)|)$, where $dom(\varepsilon) = \{el_1, el_2\}$. We abuse a notation here by using $dom(a)$ to specify the set of variables which can be assigned to value a. The variable or value which has the highest value for metric m is chosen to participate in the proposed variable-value pair for shaving. For example as presented in Figure 1, 1 is the entity with the highest value for metric m (trivially satisfied as there is only one variable/value with the domain equal 2). Therefore value 1 is retained. $dom(1)$ contains two variables $\{X_1, X_2\}$. The variable X_2 with larger domain is chosen for shaving. Therefore, for the given example, the value $(X_2, 1)$ is proposed for shaving. If there are multiple candidates among variables and values then the constraint chooses a value which produces a *ground* or *bound* event when successfully shaved. If there is no variable/value with domain consisting of two elements then alldifferent will not propose any value for shaving. In the process of selection, the preferred value is the one which, if shaved, greatly enhances the immediate propagation of the proposing constraint.

4.2 Sum

The implementation of the *sum* constraint in most constraint systems is rather simple. First, a propagator based on *bound* events is used as it provides decent propagation at low cost. It takes lower and upper bounds for all variables and checks for each bound value if there exists an assignment to other variables which satisfies the constraint. If such an assignment does not exist then the bound is tightened. This is the approach implemented in the solver used in the experiments.

The guiding function within the *sum* constraint analyzes the domains of the variables in its scope and comes up with a value within a domain of a variable which, if removed, causes the maximum amount of pruning in the other variables. The likely candidates are the domains like {1, 10..20} (resp. {1..10, 20}), where there is a large gap between the smallest (resp. the largest) element in the domain and the one which follows (resp. precedes). In this example, removing element 1 (20) from the domain will significantly tighten the bounds of a variable, probably causing some domain reductions in other variables.

Sum constraint computes for every variable X within constraint scope the following metrics, d_{min} and d_{max}. Given $dom(X) = \{v_1, v_2, ..., v_{l-1}, v_l\}$ then $d_{min} = v_2 - v_1$ and $d_{max} = v_l - v_{l-1}$. The variable X_i, for which we have $m = max(d_{min}(X_i), d_{max}(X_i))$ maximized and greater than 1, is chosen for shaving. If $d_{min}(x_i)$ is greater than $d_{max}(x_i)$ then the value (X_i, v_1) is proposed for shaving, otherwise it is value (X_i, v_l).

5 Experimental Results

In order to demonstrate the practical interest of the approach introduced in this paper, we have conducted an experimentation using different shaving ap-

proaches. We have used four metrics to compare the different approaches. They respectively correspond to the number of search nodes (# Nodes), the number of values tested for shaving (# Tests), the success shaving ratio which corresponds to the percentage of values which were shaved, and the execution time (CPU) given in seconds. For all metrics, except for the shaving ratio, we computed both average and median values (denoted by Avg and Med).

All experiments were performed on laptop with Intel Core Duo 2.0 GHz processor and 1GB of RAM running Linux Kubuntu 6.10. We have used in experiments the Java-based JaCoP solver version 2.3, which is available for free for non-commercial purposes [22]. We have considered three problem classes, namely Quasigroup Completion Problems (QCP), Nontransitive Dice Problems (NTD), and Magic Squares Problems (MSP). The different approaches are:

- NoShaving: JaCoP alone,
- QShaving: JaCoP embedding the quick shaving technique,
- GShaving: JaCoP embedding our constraint-guided shaving technique,
- GQShaving: JaCoP embedding both the quick shaving and the constraint-guided shaving techniques.

We will show that in all problem classes, we can obtain a reduction in the number of search nodes. This reduction in all problem classes translates to a time reduction. We will show that different shaving approaches complement each other.

5.1 Nontransitive Dice Problems

NTD(d,s) represents a problem involving dices. Here, d denotes the number of dice and s denotes the number of faces on each die. All faces are assumed to be different, so there is no possibility of a draw when two dice are rolled. In short, the solution to this problem assigns to each face of each die a unique value. Moreover, we are looking for an assignment of dice faces, such that for each die we can pick up another die and reach the maximum probability of winning with the first chosen die. The optimal solution for NTD(3,6) with the winning probability 21/36 is presented in Figure 2. The arrows represent the winning relation (e.g. the die on the left is winning over the die in the middle). The optimization (maximization of the minimal winning probability) is achieved by restarts with stepwise increase of the maximal probability. As soon as for a given probability no solution exists, the previous solution is proved to be optimal. By arranging experiments in such a way, we ensure that each search solves the same series of sub-problems and each search finds and proves the optimal solution.

The model of this problem is not trivial since it contains dual viewpoints, symmetry breaking constraints, and global constraints such as *sum* and *alld-ifferent*. In the process of experimentation with different search heuristics, we found the best one for JaCoP alone. This heuristic orders the face variables by taking them one by one from each die. The face variables taken from each die are ordered in the fashion which maximizes constraint propagation. In addition, this

Table 1. Experimental results for Nontransitive Dice Problem.

		# Nodes		# Tests		Shaving	CPU [s]	
Approach	#Solved	Avg	Med	Avg	Med	Ratio	Avg	Med
NoShaving	22	9,935,361	92,915	0	0	–	667	8.57
GShaving	26	1,191,156	7,650	54,302	440	49	104	3.50
QShaving	26	583,125	5,932	105,578	1,117	43	70	3.02
GQShaving	26	217,398	4,381	51,706	1,072	43	32	3.11

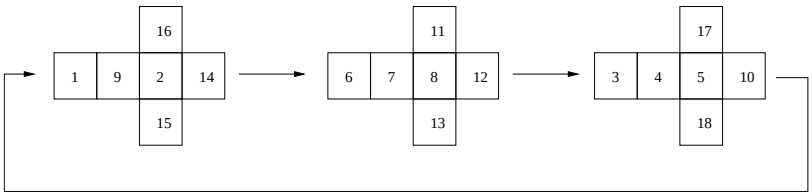

Fig. 2. NTD(3, 6) and one of the optimal solutions

heuristic uses middle value ordering which starts with values from the middle of the domain.

Table 1 presents results for 26 problem instances, namely NTD(3, {4, 5, 6, 7, 8, 9, 10}), NTD(4, {4, 5, 6, 7, 8, 9, 10}), NTD(5, {4, 5, 6, 7, 8}), NTD(6, {4, 5, 6, 7}), and NTD(7, {4, 5, 6}). JaCoP alone (NoShaving) was not able to solve 4 problems with a generous backtrack limit of 10 millions backtracks. Both shaving techniques run alone (QShaving and GShaving) already obtain substantial time gains when solving the same problem set. However, it is when both shaving techniques are combined that a 20 times reduction of average execution time is obtained. These experimental results clearly show that we are able to improve the pruning strength of the constraint model by using shaving techniques.

5.2 Quasigroup Completion Problems

A quasigroup is a set Q with a binary operation $\star : Q \times Q \rightarrow Q$, such that for all a and b in Q there exist unique elements in Q such that $a \star x = b$ and $y \star a = b$. The cardinality of the set, $n = |Q|$, is called the order of the quasigroup. A quasigroup can be viewed as an $n \times n$ multiplication table defining a Latin square, which must be filled with unique integers on each row and column. The Quasigroup Completion Problem (QCP) is the problem of completing a partially filled Latin square. We will concentrate on one of many possible constraint models which uses *alldifferent* global constraint to ensure uniqueness of elements across columns and rows. In our work, we use a generalized arc consistency (GAC) version of the *alldifferent* constraint. One of the best variable orderings, given the fact that the model consists only of global constraints, is based on the minimal domain size. We have used as value ordering heuristic the one which chooses the minimal

value from the current domain. The QCP problems were generated using Carla Gomes generator. We have generated 3000 problems for QCP of order 25 at difficulty phase transition for sat balanced QCP. The generator can produce unsat instances for this type of QCP problems therefore we retained only the sat instances which gave the total number of instances equal to 1214.

Table 2. Experimental results for QCP of order 25

		# Nodes		# Tests		Shaving	CPU [s]	
Approach	#Solved	Avg	Med	Avg	Med	Ratio	Avg	Med
NoShaving	1,103	437,321	123,070	0	0	–	264.59	74.44
QShaving	1189	212,372	47,236	88,931	19,238	32	167.90	38.33
GShaving	1,214	12,972	3,113	92,737	22,280	18	92.86	24.20
GQShaving	1,214	12,681	3,051	94,006	22,520	18	61.49	17.23

Table 2 presents experimental results for this problem class. JaCoP alone (NoShaving) had a 10 times increased search node limit yet it still could not solve 9% of the problems for which more than 1000 seconds on average were not sufficient to solve them. We were initially running experiments without any node limit. Unfortunately, we were not able to get results for some instances even within days therefore we had to revert to a node limit and present approximate results (i.e. results given for NoShaving and QShaving must be considered as lower bounds of real values). QShaving was given the same node limit as NoShaving. All remaining shaving approaches were given only one tenth of search nodes limit given to the heuristic without shaving.

The guided shaving (GShaving, GQShaving) compares favorably against NoShaving and QShaving. First, it solves all the problems in a significant smaller numbers of search nodes even if we count shaving attempts as search nodes. Assuming that unsolved instances are solved at the node limit we can still see significant reduction in median time. We run QShaving with a node limit equal to NoShaving method, because if given the same node limit as other shaving approaches it would not be able to solve 25% of the problems. QShaving takes on average more than 16 times more nodes and 80% more time when compared to GShaving results. Moreover, QShaving still despite 10-fold increase of node limit can not solve 2% of problems making it also less robust than GShaving. It is interesting to see that guided shaving significantly improves the quality of quick shaving since all problems are solved. On the other hand, quick shaving changes the distribution of shaving attempts (there are more of them in the lower parts of the search tree) as well as makes shaving attempts more compatible with constraint consistency mechanisms (reuse of previous work is more likely, the consistency function of alldifferent is highly incremental), resulting in further reduction of time even if all other metrics are similar.

5.3 MagicSquares Problems

An order n magic square is a $n \times n$ matrix containing the numbers 1 to n^2, with each row, column and the main diagonals summing up to the same number. The constraint model of this problem consists of one alldifferent constraint and $2n + 2$ sum constraints. Therefore, the guiding for this problem class is mostly performed by sum constraints.

Table 3. Experimental results for MagicSquares Problems

		# Nodes		# Tests		Shaving	CPU [s]	
Approach	#Solved	Avg	Med	Avg	Med	Ratio	Avg	Med
NoShaving	6	321,112	9,052	0	0	–	43.43	3.06
QShaving	6	56,751	8,182	8,496	920	40	10.12	3.09
GShaving	6	18,443	5,047	8,936	2,258	56	6.55	3.70
GQShaving	6	14,338	4,765	9,903	2,667	53	5.69	2.83

Table 3 presents experimental results for this problem class. We have used the variable ordering heuristic that selects the smallest domain first and the value ordering heuristic which starts with the values from the middle of the domain. The difficulty of the problems increases very fast. Therefore, only instances with small n (n starting at 4) were solved. For this problem class, we can observe a reduction in terms of the average number of search nodes as well as the average execution time. Clearly, guided shaving allows again to improve the search robustness: it is about one order of magnitude faster than the classical search algorithm (NoShaving) and two times faster than quick shaving (QShaving).

6 Conclusions

We have presented a shaving framework, which uses advice from (global) constraints. The underlying principle of constraint-guided shaving is to ask each constraint suggesting one value which is more likely to be shaved as well as cause more propagation when shaved. We have discussed how it can be implemented for two important global constraints, namely *alldifferent* and *sum*. Interestingly enough, we have also shown that the past successes and failures can be exploited to improve shaving performance.

Constraints provide reliable guidance which always allows pruning some portions of the search space while, most of the time, giving significant reduction of execution time (one order of magnitude). Indeed, the practical results that we have obtained on different series of problems show that using constraints for guiding shaving increases dramatically the robustness and the efficiency of the search algorithm, not only in terms of search nodes but also in terms of cpu time. We have also shown that our approach is complementary to Quick shaving.

References

1. Wallace, M.: Practical applications of constraint programming. Journal of Constraints **1** (1996) 139–168
2. Puget, J.: The next challenge for CP: Ease of use. Invited Talk at CP-2004 (2004)
3. Carlier, J., Pinson, E.: Adjustments of heads and tails for the job-shop problem. European Journal of Operational Research **78** (1994) 146–161
4. Martin, P., Shmoys, D.: A new approach to computing optimal schedules for the job-shop scheduling problem. In: Proceedings of IPCO'96. (1996) 389–403
5. Lhomme, O.: Quick shaving. In: Proceedings of AAAI'05. (2005) 411–415
6. Debruyne, R., Bessiere, C.: Some practical filtering techniques for the constraint satisfaction problem. In: Proceedings of IJCAI'97. (1997) 412–417
7. Hnich, B., Richardson, J., Flener, P.: Towards automatic generation and evaluation of implied constraints. Technical Report 2003-014, Uppsala Universitet (2003)
8. Bessiere, C., Coletta, R., Petit, T.: Learning implied global constraints. In: Proceedings of IJCAI'07. (2007) 50–55
9. Regin, J.: Combination of among and cardinality constraints. In: Proceedings of CPAIOR'05. (2005) 288–303
10. Debruyne, R., Bessiere, C.: Domain filtering consistencies. Journal of Artificial Intelligence Research **14** (2001) 205–230
11. Bessiere, C., Stergiou, K., Walsh, T.: Domain filtering consistencies for non-binary constraints. Artificial Intelligence, to appear (2008)
12. Bessiere, C.: Constraint propagation. In: Handbook of Constraint Programming. Elsevier (2006)
13. Bartak, R., Erben, R.: A new algorithm for singleton arc consistency. In: Proceedings of FLAIRS'04. (2004)
14. Bessiere, C., Debruyne, R.: Optimal and suboptimal singleton arc consistency algorithms. In: Proceedings of IJCAI'05. (2005) 54–59
15. Lecoutre, C., Cardon, S.: A greedy approach to establish singleton arc consistency. In: Proceedings of IJCAI'05. (2005) 199–204
16. Lecoutre, C., Prosser, P.: Maintaining singleton arc consistency. In: Proceedings of CPAI'06 workshop held with CP'06. (2006) 47–61
17. Hwang, J., Mitchell, D.: 2-way vs d-way branching for CSP. In: Proceedings of CP'05. (2005) 343–357
18. Sabin, D., Freuder, E.: Contradicting conventional wisdom in constraint satisfaction. In: Proceedings of CP'94. (1994) 10–20
19. Bessiere, C., Zanuttini, B., Fernandez, C.: Measuring search trees. In: Proceedings of ECAI'04 workshop on Modelling and Solving Problems with Constraints. (2004) 31–40
20. Schulte, C., Carlsson, M.: Finite domain constraint programming systems. In: Handbook of Constraint Programming. Elsevier (2006) 495–526
21. Regin, J.: A filtering algorithm for constraints of difference in CSPs. In: Proceedings of AAAI'94. (1994) 362–367
22. Kuchcinski, K.: Constraints-driven scheduling and resource assignment. ACM Transactions on Design Automation of Electronic Systems **8** (2003) 355–383