International Journal on Artificial Intelligence Tools
Vol. 17, No. 4 (2008) 703–730
© World Scientific Publishing Company

World Scientific
www.worldscientific.com

A SPACE-EFFICIENT BACKTRACK-FREE REPRESENTATION FOR CONSTRAINT SATISFACTION PROBLEMS

J. CHRISTOPHER BECK

Department of Mechanical & Industrial Engineering, University of Toronto
5 King's College Rd., Toronto, Ontario, M5S 3G8, Canada
jcb@mie.utoronto.ca

TOM CARCHRAE

Actenum Corporation, Suite 410 – The Landing
375 Water Street, Vancouver, British Columbia, Canada
carchrae@actenum.com

EUGENE C. FREUDER

Cork Constraint Computation Centre
University College Cork, Cork, Ireland
e.freuder@4c.ucc.ie

GEORG RINGWELSKI

Computer Science Department, Hochschule Zittau/Goerlitz
University of Applied Sciences
Brueckenstrasse 1, 02826 Goerlitz, Germany
gringwelski@hs-zigr.de

Received 30 March 2007
Accepted 13 February 2008

In this paper we present a radical approach to obtaining a backtrack-free representation for a constraint satisfaction problem: remove values that lead to dead-ends. This technique does not require additional space but has the drawback of removing solutions. We investigate a number of variations on the basic algorithm including the use of seed solutions, consistency techniques, and a variety of pruning heuristics. Our experimental results indicate that a significant proportion of the solutions to the original problem can be retained especially when an optimization algorithm that specifically searches for such "good" backtrack-free representations is employed. Further extensions increase solution retention by searching for high-coverage backtrack-free representations, by removing tuples rather than values, and by combining multiple backtrack-free representations. Our approach elucidates, for the first time, a three-way trade-off between space complexity, potential backtracks, and solution loss and enables algorithms that can actively reason about the trade-off between space, backtracks, and solution loss.

Keywords: Backtrack-free search; space efficient; constraint satisfaction; consistency processing.

1. Introduction

The two fundamental techniques for solving constraint satisfaction problems (CSPs) are search and inference. In the former, the central approach is to assign values to variables, backtracking to try other assignments when it is discovered that the current set of assignments cannot lead to a solution. In the latter, reasoning is done on the constraint representation to remove values from variable domains that can be shown not to participate in a solution. Pure versions of either approach can be used to solve CSPs, however, in practice these techniques are combined by doing low levels of inference in states produced from variable assignments.

We are interested in offline/online problems: problems that will be solved many times in an online context where minimization of backtracking is necessary but for which there is the luxury of significant offline processing time. A standard technique in such an application is to compile the constraint problem into some form that allows backtrack-free access to solutions.[17] However, except in special cases, the worst-case size of the compiled representation is exponential in the size of the original problem. Therefore, the common view of the dilemma is as a trade-off between storage space and backtracks: worst-case exponential space requirements can guarantee backtrack-free search (e.g., through adaptive consistency[5] techniques with a fixed maximum constraint arity) while polynomial space techniques leave the risk of backtracking.

In this paper, we suggest that viewing the problem as the two-way trade-off is an unnecessarily limited perspective and that there exists a three-way trade-off among storage space, backtracking, and solution retention. In the extreme, we propose a simple, radical approach to achieving backtrack-free search: preprocess the problem to remove *values* that lead to dead-ends. For example, consider a coloring problem with variables $\{X, Y, Z\}$ and colors $\{red, blue\}$. Suppose Z must be different from both X and Y and our variable ordering is lexicographic. There is a danger that the assignments $X = red$, $Y = blue$ will be made, resulting in a domain wipe-out for Z. A conventional way of fixing this would be to add a new constraint between X and Y specifying that the tuple in question is prohibited. Such a constraint requires additional space, and, in general, these consistency methods may have to add constraints involving as many as $n - 1$ variables for an n-variable problem. Our basic insight here is simple, but counter-intuitive. We will "fix" the problem by removing the choice of red for X. One solution, $\{\langle X = red\rangle, \langle Y = red\rangle, \langle Z = blue\rangle\}$, is also removed but another remains: $\{\langle X = blue\rangle, \langle Y = blue\rangle, \langle Z = red\rangle\}$. If we also remove red as a value for Y we are left with a backtrack-free representation for the problem.[a] This extreme example, where values are removed, allows us to achieve a "backtrack-free" representation (BFR) where all *remaining* solutions can be enumerated without backtracking and where the space complexity is the same as

[a]In this case, but not always, the representation also leaves us with singleton domains for each variable.

for the original CSP. We are able to achieve backtrack-free search *and* polynomially bounded storage requirements at the cost of some solution loss.

More generally, we define a family of approaches called k-BFR where k is the maximum arity of constraints that can be added to remove dead-ends. Removing values is 1-BFR, as value removals can be represented as the addition of unary not-equals constraints. At the other end of the spectrum n-BFR (where n is the number of variables in the problem) is equivalent to adaptive consistency: no solutions are lost but space requirements may be exponential in the problem size.[4] Between these extremes are a number of intermediate points that allow the addition of new constraints up to a fixed arity. The choice of k therefore allows storage space to be directly traded-off against solution loss.

In the next section, we informally present the core idea of 1-BFR and an example CSP that is used through-out the paper. In Section 3, we provide a more detailed motivation and review related work. In Section 4, we look at two simple extensions to the basic algorithm that make it much more efficient and also trivial to show correctness. Section 5 presents the pseudocode for the extended algorithm and a complexity analysis. In Section 6, we perform experimental studies of the 1-BFR algorithm to look at the number and quality of the solutions that are retained and the effect of heuristics. Section 7 investigates extensions of 1-BFR: maintaining multiple 1-BFR representations for the same problem instance and moving to k-BFR and another variation, restricted BFR. Section 8 returns to the central issue of solution loss, discusses how variations in the online algorithm impact the offline processing, and links this work to existing research. Finally, in an appendix we present the general 1-BFR algorithm and prove its correctness.

2. Creating a 1-BFR: The Core Idea

A constraint satisfaction problem (CSP), P, is a triple (V, D, C) where V is a set of variables, D is a set of domains $D_i \in D$ such that D_i is the set of possible values to which variable V_i can be assigned, and C is a set of constraints defining the tuples of values to which subsets of the variables can be assigned. Given a problem P and a static variable search order V_1 to V_n, we will refer to the subproblem induced by the first k variables as P_k. Note that problem P_k includes V_k. A variable V_i is a parent of V_k if it shares a constraint with V_k and $i < k$. We call the subproblem induced[b] by the parents of V_k, PP_k, the *parent (sub)problem* of V_k. P_n is a backtrack-free representation if any choice of values under the variable ordering for V_1 to V_n leads to a solution without backtracking.

The basic 1-BFR algorithm is quite straightforward, largely inspired by strong directional k-consistency.[6,5] Starting with the last variable, V_n, all consistent tuples of the parent subproblem are enumerated. For each such tuple, t, V_n is examined to

[b]Note that the number of variables in the parent subproblem is bounded by the induced width of the variable search order.

see if some value in its domain is consistent with t. If such a value exists, nothing is done. If no values in the domain of V_n are consistent with t then one of the values in t is removed. More formally, if $t = (\langle V_i = x_i \rangle, \langle V_j = x_j \rangle, \ldots, \langle V_{n-1} = x_{n-1} \rangle)$, the value x_k will be removed from the domain of the variable V_k for some $\langle V_k = x_k \rangle \in t$. Such a removal means that during online search the tuple t will never be found and therefore the dead-end will not occur, due to the fact that t cannot be extended to V_n. After all such tuples for the parent subproblem of V_n are processed, the algorithm is recursively applied to V_{n-1}. It is possible with this algorithm that the removal of $\langle V_k = x_k \rangle \in t$ results in V_k having an empty domain. In such a case, the algorithm must change some of its previous domain removals.

In the Appendix we show that this algorithm is guaranteed to find a backtrack-free representation for any soluble problem. In the rest of this section, we illustrate the algorithm on the following small example CSP that is used through-out the paper:

- Variables: V_1, V_2, V_3, V_4
- Initial domains: $D_i = \{1..10\}$
- Constraints:
  (1) $\mathit{alldifferent}([V_1, V_2, V_3, V_4])$
  (2) $V_1 + V_2 = 7$
  (3) $V_1 - V_3 > V_2$

We assume a static variable ordering with ascending index: $V_1, \ldots, V_4$. The iterations of the basic 1-BFR algorithm on this problem instance are as follows:

1. In the first iteration, we look at the parent subproblem of V_4: $(\{V_1, V_2, V_3\}, \{D_1, D_2, D_3\}, \{V_1 + V_2 = 7,\ V_1 - V_3 > V_2\})$. There are two solutions to this subproblem which do not extend to V_4 because constraint (1) is not satisfied: $(\langle V_1 = 5 \rangle, \langle V_2 = 2 \rangle, \langle V_3 = 2 \rangle)$ and $(\langle V_1 = 6 \rangle, \langle V_2 = 1 \rangle, \langle V_3 = 1 \rangle)$. The basic 1-BFR algorithm chooses one value removal for each of these tuples. In our example, assume that the values $\langle V_1 = 5 \rangle$ and $\langle V_1 = 6 \rangle$ are removed.
2. In the next iteration, the algorithm examines the parent subproblem of V_3 which, with the domain reductions from the first iterations, is: $(\{V_1, V_2\}, \{D_1, D_2\}, V_1 + V_2 = 7)$ with $D_1 = \{1, 2, 3, 4, 7, 8, 9, 10\}$. It has the solutions $\{(\langle V_1 = 1 \rangle, \langle V_2 = 6 \rangle), (\langle V_1 = 2 \rangle, \langle V_2 = 5 \rangle), (\langle V_1 = 3 \rangle, \langle V_2 = 4 \rangle), (\langle V_1 = 4 \rangle, \langle V_2 = 3 \rangle)\}$, none of which extends to V_3 due to constraint (3). Thus the algorithm could prune the values $\{1, 2, 3, 4\}$ from V_1 leading to $D_1 = \{7, 8, 9, 10\}$.
3. Moving to V_2, the parent subproblem contains only V_1 and the values $\{7, 8, 9, 10\}$ must be pruned from V_1 as none of them is consistent with any values in D_2: all values fail to satisfy constraint (2). This will lead to $D_1 = \emptyset$.
4. The BFR algorithm would thus "backtrack" and re-consider the parent subproblem of V_3. All of the possible value removals eventually result in a domain wipe-out. Once this is discovered through a series of prunings and backtracks, the algorithm backtracks further to V_4.

5. In considering again the solutions to the parent subproblem of V_4, neither $(\langle V_1 = 5\rangle, \langle V_2 = 2\rangle, \langle V_3 = 2\rangle)$ nor $(\langle V_1 = 6\rangle, \langle V_2 = 1\rangle, \langle V_3 = 1\rangle)$ can be extended to V_4. The algorithm could now choose to prune the values $\langle V_3 = 1\rangle$ and $\langle V_3 = 2\rangle$.
6. The algorithm can now again move forward, addressing the parent subproblem of V_3: $(\{V_1, V_2\}, \{D_1, D_2\}, V_1 + V_2 = 7)$. This problem has six solutions: $\{(\langle V_1 = 1\rangle, \langle V_2 = 6\rangle), (\langle V_1 = 2\rangle, \langle V_2 = 5\rangle), (\langle V_1 = 3\rangle, \langle V_2 = 4\rangle), (\langle V_1 = 4\rangle, \langle V_2 = 3\rangle), (\langle V_1 = 5\rangle, \langle V_2 = 2\rangle), (\langle V_1 = 6\rangle, \langle V_2 = 1\rangle)\}$. Only the final solution will extend to V_3 due to constraint (3) and so the algorithm must choose values to prune. Assume that the values $V_1 = \{1, \ldots, 5\}$ are removed. Leaving $D_1 = \{6, 7, 8, 9, 10\}$.
7. Moving again to the parent subproblem of V_2, the values $\{7, 8, 9, 10\}$ of V_1 must be removed (as above). However, as $\langle V_1 = 6\rangle$ is still a valid assignment, the algorithm terminates producing the following BFR: $D_1 = \{6\}, D_2 = \{1 \ldots 10\}, D_3 = \{3, \ldots, 10\}, D_4 = \{1 \ldots 10\}$. This BFR covers 14 of the 28 solutions to the problem.

A number of the details of our approach to the creation of backtrack-free representations are illustrated by this example. First, notice that during online search a backtrack is when search must return to a previously assigned variable and change the assignment. We do not consider a backtrack to have occurred if we can immediately detect that a value for the to-be-assigned variable is inconsistent based on checking against the assignments that have already been made. Second, as with strong directional k-consistency, we assume a static variable ordering is known and will be used in the online solving. Third, the need to change previous pruning decisions as well as the number of solutions that are ultimately retained depend on the non-deterministic pruning decisions. We have not yet specified how these pruning decisions are made and experiment with a number of heuristics below.

3. Background

Having presented the basic idea behind BFRs, in this section, we turn to our motivation for this work as well as the constraint literature that this work is built upon.

3.1. *Motivation*

One offline/online application arises in online configuration where, via a web interface, a customer selects particular options for some consumer product (e.g., a car, a camera) in order to customize the product to his or her requirements and preferences. For example, on the Renault website (www.renault.de), the customer is prompted for certain decisions, in order, such as the basic car model, diesel or regular engine, etc. The decisions have impact on subsequent options that are available and the overall price. However, the eventual impact of a particular decsion is not clear at the time of the decision-making. If it is later realized that the car is now too expensive, the user must use the “back” button on the browser to return to earlier

decisions. The Porsche configurator (www.porsche.de) goes one step further in using pop-up windows to inform the user about the consequences of decisions. However, as the problem of guaranteeing that the user will never encounter a dead-end is NP-complete, we do not believe the Porsche website provides such a guarantee.

Given human impatience, backtracking in such a context detracts from the user experience and is likely to lead the user to abandon the interaction. Our basic approach in this paper is to investigate a mechanism to guarantee that backtracking cannot happen, at the cost of making some configurations impossible. From a product design perspective, businesses often decide that some product configurations will not be sold for mechanical, usability, production, financial, or less tangible reasons such as reputation. For example, the user interface on a digital camera may become too cumbersome if all the mechanically possible options are selected. As such an interface will reflect poorly on the design reputation of the firm, particular combinations of options are disallowed. Our approach suggests that businesses may also be willing to remove certain configurations in order to automate the sales process. As with other considerations, the tangible and intangible benefits arising from such a streamlined sales process may more than out-weigh the costs of limiting a product's configuration space.

While, in practice, we believe a business will want to be more nuanced in their ability to trade-off a backtrack-free sales interaction against product restrictions, for clarity, we present the work in its most basic and radical form. Subsequently, in Section 8.2, we demonstrate how our backtrack-free representations can be used to deal more subtly with such a trade-off.

3.2. *Related work*

Early work on CSPs guaranteed backtrack-free search for tree-structured problems.[8] This was extended to general CSPs through k-trees[9] and adaptive consistency.[5] These methods have exponential worst-case complexity, but, for preprocessing, time is not a critical factor as we assume we have significant time offline. However, these methods also have exponential worst-case space complexity, which may indeed make them impractical.

Efforts have been made to precompile all solutions in a compact form.[1,11,12,14,17] These approaches achieved backtrack-free search but also at the cost of worst-case exponential storage space. While a number of interesting techniques to reduce average space complexity (e.g., meta-CSPs and interchangeability[17]) have been investigated, they do not address the central issue of worst-case exponential space complexity. Indeed, as far as we have been able to determine, the need to represent all solutions has not been questioned in existing work in consistency processing for backtrack-free search.

Figure 1 presents a spatial representation of a number of standard CSP search techniques, adaptive consistency, and solution compilation approaches together with the work in this paper. We believe that the figure provides a novel perspective

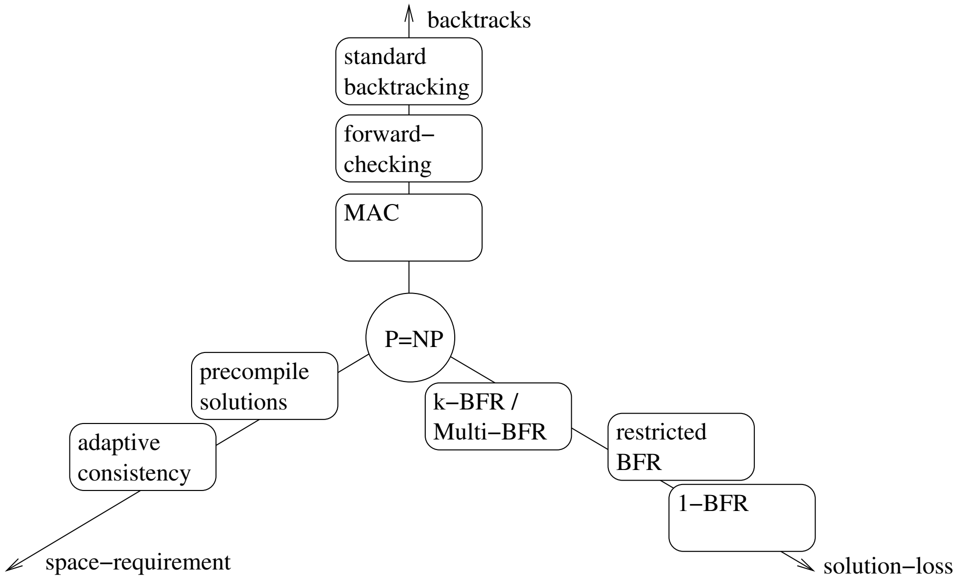

Fig. 1. A three dimensional view of CSP algorithms.

on these techniques on the three important dimensions that we identified above: number of backtracks, space complexity, and solution loss. Standard search techniques, for example, have focused on the dimension of backtracks: higher levels of consistency and heuristic search methods (not shown) strive to reduce the number of backtracks while maintaining all solutions and polynomial-space complexity. Adaptive consistency and the compilation methods seek to maintain all solutions and remove all backtracks, while incurring a worst case exponential space complexity. Finally, the techniques growing from the core idea of 1-BFR and defined below investigate the solution loss axis: trading solution retention for zero backtracks and polynomial space complexity.

4. Simple Extensions to the Core Idea

While we are assuming that the creation of a BFR is done offline and therefore we have considerable time in which it may be done, it is clear that the run-time of the basic 1-BFR algorithm can be improved in a number of ways. For example, two consecutive sub-problems (i.e., PP_k and PP_{k-1}) are likely to share variables. Given that the basic algorithm enumerates each possible parent tuple and detects if it can be extended to the child, we may be able to share information between sub-problems and so reduce the offline solving time. Two different extensions are investigated in this section: enforcing consistency whenever we remove a value and protecting the values in a *seed* solution to ensure that no domain will become empty during the creation of a 1-BFR. We leave the sharing of information between sub-problems for future work.

Consistency Enforcement. Given the power of consistency enforcement in standard CSP solving, we expect it will both reduce the effort in searching for a BFR and reduce the pruning decisions that must be made. Whenever a value is removed from a domain, we can perform constraint propagation on the entire problem to establish the desired level of consistency. In our example and experiments (Section 6), we enforce arc consistency (AC). Other levels of consistency (e.g., bounds consistency[13]) will also be useful depending on the arity and underlying complexity of the propagation algorithms for the constraints in the problem.

Our preliminary experiments showed that establishing AC on the whole problem (i.e, P_n) whenever a value is pruned reduces the computational effort to find a BFR and results in BFRs which represented more solutions.

Seed Solutions. The basic algorithm for finding a 1-BFR suffers from the possibility of creating an empty domain for a variable in a parent problem and, therefore, having to revise previous pruning decisions. We can guarantee that the algorithm will never empty a variable's domain by creating a BFR around a "seed" solution. We either use standard CSP search techniques to find a solution to the problem or use a solution provided to us from some other source (e.g., a preferred solution from the system designer[c]). Then we modify the value pruning to specify that non-extendible parent solutions must be removed by pruning values that do not appear in the seed solution. This is sufficient to guarantee that there will never be a need to undo pruning decisions: the seed solution itself is a BFR and so if we never remove values in the seed solution, the algorithm will terminate, in the worst case, returning the seed solution itself.

We perform a series of experiments below on the usefulness of seed solutions and techniques to find seed solutions that will lead to BFRs that represent many solutions.

Example. To give an idea of the impact of using seed solutions and enforcing arc consistency, we now return to our example problem. Assume that the solution $(\langle V_1 = 6\rangle, \langle V_2 = 1\rangle, \langle V_3 = 3\rangle, \langle V_4 = 2\rangle)$ is our seed. A BFR can now be found as follows:

1. We run arc consistency on the initial problem resulting in the following domains: $D_1 = \{3, 4, 5, 6\}$, $D_2 = \{1, 2, 3, 4\}$, $D_3 = \{1, 2, 3, 4\}$, and $D_4 = \{1, \ldots, 10\}$.
2. In the first iteration, the same parent solutions as above cannot be extended: $(\langle V_1 = 5\rangle, \langle V_2 = 2\rangle, \langle V_3 = 2\rangle)$ and $(\langle V_1 = 6\rangle, \langle V_2 = 1\rangle, \langle V_3 = 1\rangle)$. Above, we removed $\langle V_1 = 5\rangle$ and $\langle V_1 = 6\rangle$. Here we remove $\langle V_1 = 5\rangle$ but the second removal is no longer allowed as $\langle V_1 = 6\rangle$ is part of the seed solution. To remove the second non-extending parent solution we remove $\langle V_3 = 1\rangle$. Arc consistency propagation on the whole problem then removes $\langle V_2 = 2\rangle$, $\langle V_2 = 4\rangle$, and $\langle V_1 = 3\rangle$.

[c] Another important benefit of using a seed solution, is therefore, the ability to preserve a specific solution that the system designer wants to be present.

3. In the next iteration, the algorithm examines the parent subproblem of V_3 which is: $(\{V_1, V_2\}, \{D_1, D_2\}, V_1 + V_2 = 7)$ with $D_1 = \{4, 6\}$ and $D_2 = \{1, 3\}$. It has the solutions $\{(\langle V_1 = 4\rangle, \langle V_2 = 3\rangle), (\langle V_1 = 6\rangle, \langle V_2 = 1\rangle)\}$. The latter solution is extendible so we need to prune values to remove the former. We remove the value $\langle V_1 = 4\rangle$. Enforcing arc consistency on the whole problem subsequently removes $V_2 = \{3\}$ and $V_4 = \{1, 6\}$.
4. Finally, the parent subproblem of V_2 is processed. V_1 has one possible value and it is consistent with the only value of V_2. No pruning is necessary and the algorithm terminates with the following BFR: $D_1 = \{6\}$, $D_2 = \{1\}$, $D_3 = \{2, 3, 4\}$, and $D_4 = \{2, 3, 4, 5, 7, 8, 9, 10\}$. This BFR covers 21 of the 28 solutions to the problem. Recall that without the seed solution and propagation only 14 solutions were retained. The additional solutions arise from the fact that $\langle V_3 = 2\rangle$ remains in the BFR.

5. Algorithm and Analysis

There is clearly an extra offline computational cost in using a seed solution (i.e., the need to obtain the seed) and preserving a seed reduces the flexibility we have in choosing which values to remove during the creation of the BFR. However, our preliminary experiments showed that using a seed is significantly faster than not using one. Furthermore, the BFRs that were produced based on a seed solution tended to represent more solutions than those created without a seed. In addition, the seeded algorithm is considerably easier to present and trivial to prove correctness. As a consequence, in this section, and the balance of the paper, we concentrate solely on producing BFRs using seeds. In the Appendix A we provide a general formulation of a 1-BFR algorithm that does not use seed solutions and prove its correctness.

The algorithm to find a 1-BFR using some level of consistency enforcement and a seed solution is shown in Algorithm 1. The use of the seed solution makes the correctness of the algorithm trivial as it relies on the well-established correctness of standard CSP solving techniques. We do not, therefore, provide a formal proof, but argue informally as follows. The only place the algorithm can fail is at line 7, if it is impossible to choose a such that $a \notin s^*$. To see that this cannot happen, assume that up to some value of $i = k$, we have been successful at removing values that did not appear in the seed solution. At $i = k$ assume there is a solution, s, to PP_k that cannot extend to V_k and all the assignments $a \in s$ also appear in s^*. Since we have not removed any assignments in s^*, the value of V_k in s^* must still be in D_k. Since $s \subseteq s^*$, s must be extendible to V_k by choosing the value that appears in s^*. This contradicts our assumption that s could not be extended to V_k. Note that a failure at line 9, during consistency enforcement, is impossible as we have not removed any values in the seed solution and therefore at least one value in each domain is globally consistent.

The space complexity of 1-BFR is polynomial in the number of variables and values: we are only required to represent the domains of each variable. The algorithm

Algorithm 1 Seeded-1-BFR - finds a 1-BFR using a seed solution

Seeded-1-BFR(n, s^*)

1: **if** s^* is NIL **then**
2: return Failure
3: **end if**
4: enforce consistency on P_n
5: **for** i $\leftarrow$ n to 2 **do**
6: **for all** solutions s to PP_i that do not extend to V_i **do**
7: choose $a = (V_j, x_j)$ s.t. $a \in s$ and $a \notin s^*$
8: remove x_j from D_j
9: enforce consistency on P_n
10: **end for**
11: **end for**
12: return Success

Obtains a 1-BFR for a CSP using seed

requires enumerating the solutions to the parent problem of $n - 1$ variables. The worst-case time complexity is exponential in $(p+1)$, where p is the size of the largest parent subproblem. Of course, $p + 1 = n$ in the worst case.

6. Basic Experiments

In this section, we present an initial set of experiments with the Seeded-1-BFR algorithm. Our main interest in terms of computational effort is the cost of the offline processing required to find a BFR. Given that, except for very easy problems, the BFR will contain fewer solutions than the original problem, we then turn to measurements of the number and quality of solutions that are retained. Finally, we examine if the standard CSP variable ordering heuristics can be used as pruning heuristics to produce BFRs that maintain more and better solutions. Unless otherwise noted, pruning decisions are made by randomly choosing to remove one of the non-seed values in the non-extendible parent solution.

6.1. *Computational effort*

While we are assuming that BFRs will be generated offline, an understanding of the computational effort is still important to assess the basic feasibility of the approach. The online behavior is also important, however, in a BFR, all remaining solutions can be enumerated in linear time in the number of solutions. As linear time is optimal, empirical analysis does not seem justified. Similarly, the (exponential) behavior of finding solutions in a standard CSP is well-known.

We generated random binary CSPs specified with 4-tuples (n, m, d, t), where n is the number of variables, m the size of their domains, d the density (i.e., the proportion of pairs of variables that have a constraint between them) and t the

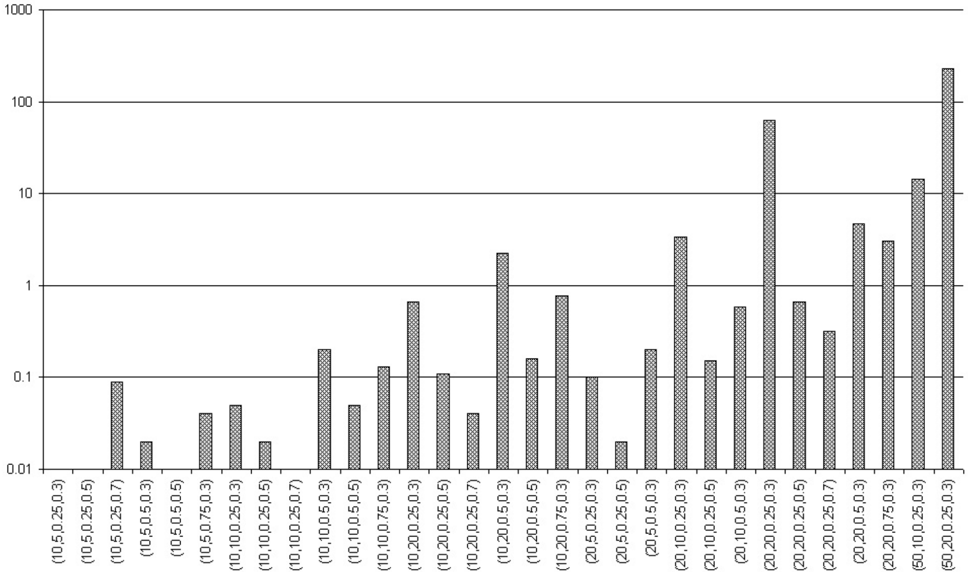

Fig. 2. Average runtime (seconds) to produce a BFR using Seeded-1-BFR across different CSP configurations (n, m, d, t).

tightness (i.e. the proportion of inconsistent pairs in each constraint). We generated at least 50 problems for each tested CSP configuration but removed insoluble instances leaving at least 30 instances from each configuration. In the following we refer to the mean of those 30 to 50 instances. Figure 2 presents the CPU time for the problems considered in our experiments *including the time to compute a seed solution*. The algorithm was implemented in C++ on a 1.8GHz Pentium 4 with 512MB of memory. Times were recorded under Windows 2000. It can be seen that the time to find BFRs scales well enough to produce them easily for the larger problems of our test set.

6.2. *Solution coverage*

Using problems generated as above with the configuration $(15, 10, 0.1, t)$, Figure 3 presents the absolute and relative solution retention as the tightness, t, is varied. The values for $t \leq 0.4$ are estimated from the portion of solutions on the observed search space and the size of the non-observed search space. Experiments with fewer samples revealed similar patterns for smaller $(10, 5, 0.5, t)$ and larger $(50, 20, 0.7, t)$ problems.

While the absolute number of solutions retained naturally decreases when the problems become tighter (there are fewer solutions and they are harder to represent in a single BFR), the relative number of solutions retained decreases up to $t = 0.6$ and increases thereafter. This increase can be explained by the fact that a BFR always represents at least one solution. The tighter problems have very few

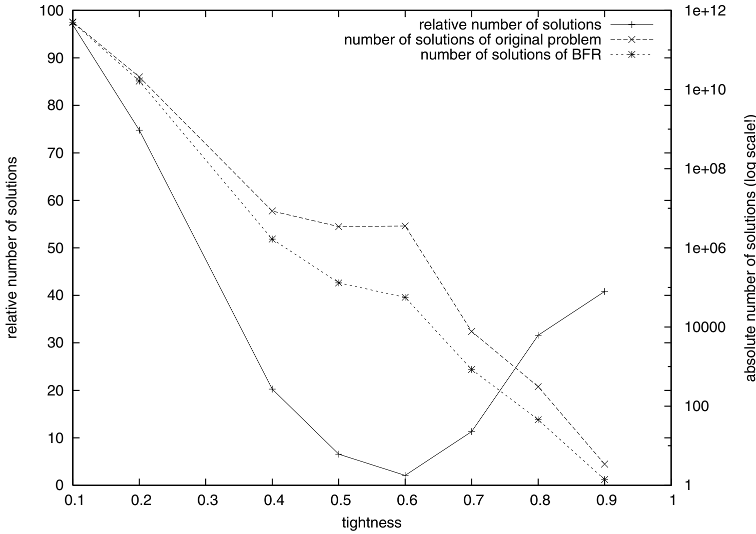

Fig. 3. Absolute and relative number of solutions retained in the BFRs produced by Seeded-1-BFR.

solutions, and therefore even retaining one solution represents a substantial portion of the total number of solutions. In contrast, for very easy problems, there may not be much need to backtrack and thus to prune when creating the BFR. In the extreme case, the original problem is already backtrack-free and therefore 100% solution retention is achieved. We observed the decreasing/increasing behavior for a range of density and tightness values. In Figure 4, we show the relative solution retention for $(15, 10, d, t)$ problems, where $d \in \{0.4, \ldots, 1\}$ and $t \in \{0.1, \ldots, 1\}$ both in steps of 0.1.

6.3. *Solution quality*

In applications where all satisfying solutions are not equally preferred, it is desirable to generate BFRs that retain good solutions. To investigate this, we examine a set of lexicographic CSPs[7] where the solution preference is expressed via a re-ordering of variables and values such that lexicographically smaller solutions are preferred.

To generate the 1-BFR, a seed solution was found using lexicographic variable and value ordering heuristics that ensures that the first solution found is optimal. The best solution will thus be protected during the creation of the BFR and will always be represented. For the evaluation of the BFR, we used the set of its solutions or a subset of it that could be found within a time limit. In Figure 5, we present the number of solutions and their lexicographic rank for $(10, 5, 0.25, 0.7)$ problems. The problem instances are shown on the x-axis, sorted in increasing number of solutions. The solutions are presented on the y-axis with increasing quality. Every block represents a solution retained by the BFR for this problem instance. In the figure we can observe for example, that instance 35 has 76 solutions and its BFR has

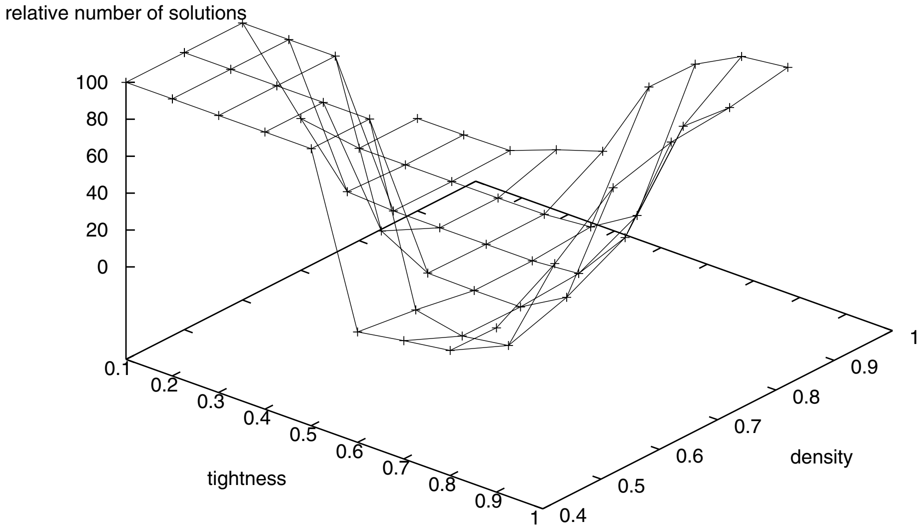

Fig. 4. Percentage of solutions retained by the BFRs produced by Seeded-1-BFR on problem instances with differing density and tightness.

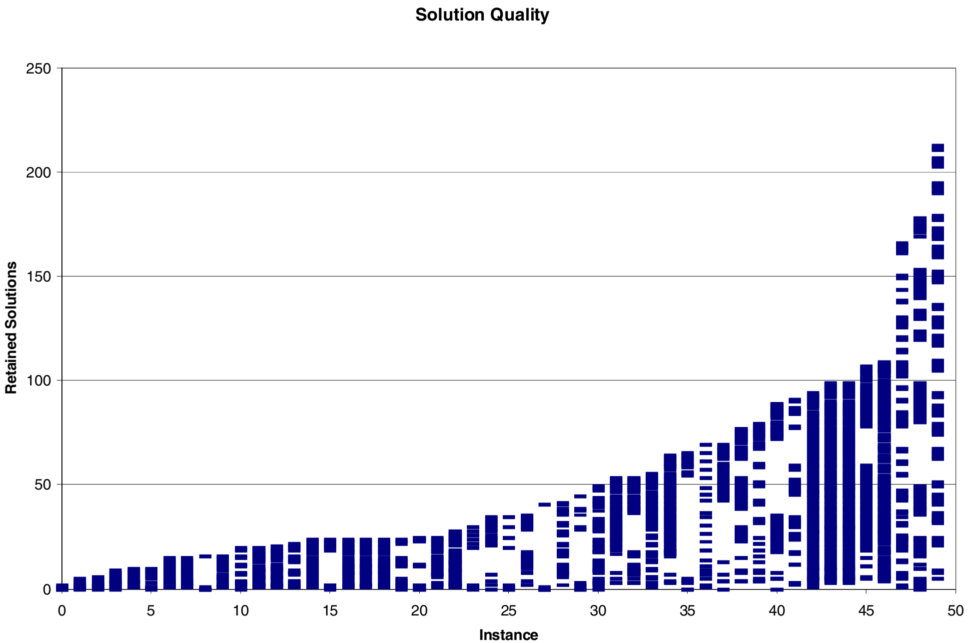

Fig. 5. Solutions maintained by a BFR generated by Seeded-1-BFR with min-degree heuristic for $(10, 5, 0.25, 0.7)$ problems.

a cluster of very high quality and a smaller cluster of rather poor quality solutions. On average, we retain over 80% of the solutions at or above the 90-percentile across these problem instances. On our worst instance, we retain only 20% of these solution but on over half of the instances, we retain all solutions in the 90-percentile and above.

These solutions were generated using a min-degree heuristic for choosing the variable whose value is pruned (i.e., line 7 of the Seeded-1-BFR algorithm). We discuss such heuristics in the following section.

The use of a lexicographic optimization function is, in retrospect, likely to be advantageous for our purposes. The high quality solutions will tend to have many variable assignments in common and will therefore be retained in our backtrack-free representation. While solution clustering has been observed in optimization problems (e.g., the existence of backbones in job shop scheduling optimization problems[16]), it seems likely that if high quality solutions are very diverse, the performance of our BFR in terms of solution quality will suffer. An interesting aspect of future research toward the practical applications of this work is in determining the extent of diversity among high-quality solutions in a given domain and whether the degradation in performance that we expect is manifest.

6.4. *Pruning heuristics*

With the importance of variable and value ordering heuristics for standard CSPs, it seems reasonable that the selection of the value to be pruned may benefit from heuristics. More precisely, at line 7 of Algorithm 1, it is necessary to choose one of the variable/value pairs (V_j, x_j) in the non-extendible parent solution and to remove the x_j from the domain of variable V_j. Ideally, we would like to choose globally inconsistent values: values that do not appear in any solution. Failing that we prefer to remove values that take part in the fewest solutions and the most non-solutions.

It is not clear, however, how the standard CSP heuristics transfer to pruning heuristics in Seeded-1-BFR. In particular, because we are choosing a variable/value pair we could base our intuitions either on traditional variable ordering heuristics (e.g., a value in the minimum domain is more likely to participate in a larger proportion of both the remaining dead-ends and solution than a value in a larger domain) or on traditional value ordering heuristics (e.g., remove the value with the the lowest promise[10] as such a value is least likely to participate in any solutions). In this paper, we restrict our experiments to techniques motivated by traditional variable ordering heuristics.

To investigate this point, we examined the following heuristics:

- Domain size: remove a value from the variable with minimum or maximum current domain size. We define the current domain of a variable to be its initial domain in the problem definition less any domain prunings performed by the Seeded-1-BFR at lines 4, 8, or 9 (see Algorithm 1).

- Degree: remove a value from the variable with maximum or minimum degree. The degree of a variable is its static degree in the problem definition.
- Lexicographic: given the lexicographic preference on solutions, remove low values from important variables, in two different ways: (1) prune the value from the lowest variable whose value is greater than its position or (2) prune any value that is not among the best 10% in the most important 20% of all variables.
- Random: remove a value from a randomly chosen variable.

If the heuristically preferred value occurs in the seed solution, the next most preferred value is pruned.

BFRs were found with each of the seven pruning heuristics. Using a set of 1600 problems with varying tightness and density, we observed little difference among the heuristics: none performed significantly better than random on either number or quality of solutions retained. We, therefore, do not present any graphical representation of our results.

Apparently, our intuitions from standard CSP variable-ordering heuristics are not directly applicable to finding good BFRs. It may be that the use of variable-ordering heuristics is fundamentally misguided as the basic decision is to choose a variable-value *pair*. Ideally, we want to remove the pair that takes part in the fewest solutions and the most non-solutions and so one possibility is to investigate the inverse of Geelen's promise heuristic (i.e., choose the pair with the most conflicts with values in the domains of neighboring variables).[10] Further work is necessary to understand the behavior of these heuristics and to determine if other heuristics can be applied to significantly improve the solution retention and quality of the BFRs. Empirical experimentation similar to work on variable-ordering heuristics[2,15] but aimed at understanding why a given variable-value pair is a good choice is a necessary step to designing a well-founded heuristic.

7. Extensions and Further Experiments

In this section, we investigate extensions of the basic Seeded-1-BFR algorithm. First we examine the usefulness of viewing the search for a 1-BFR as an optimization problem where we want to maximize the number of solutions that are retained. We investigate a simple approach called *probing*, based on generating different seed solutions, corresponding BFRs, and keeping the BFR with the highest solution coverage. Second, we investigate the feasibility of finding and maintaining multiple BFRs for the same problem.

7.1. *Probing*

Since we want BFRs to represent as many solutions as possible, it is useful to model the finding of BFRs as an optimization problem rather than as a satisfaction problem. There are a number of ways to search for a good BFR, for example, by placing Seeded-1-BFR within a branch-and-bound to find the BFR with the

maximal number of solutions given all possible sets of prunings. We investigate a simpler technique: blind probing. Because we generate BFRs starting with a seed solution, we can iteratively generate seed solutions and corresponding BFRs and keep the BFR that retains the most solutions. For a given seed solution, a BFR is generated using random pruning decisions and the number of solutions retained is counted. Using a random variable and value ordering, we then generate another seed solution to produce a corresponding BFR. At each stage we count the solutions retained and preserve the BFR with the highest coverage. This process is continued until no improving BFR is found in 1000 consecutive iterations. Probing is incomplete: it is not guaranteed to find the BFR with maximal coverage. However, not only does such a technique provide significantly better BFRs based on solution retention, it also provides a baseline against which to compare our satisfaction-based BFRs.

Table 1 presents the mean number of solutions retained using random pruning with and without probing on seven different problem sets each with 50 problem instances. Probing is almost always able to find BFRs with higher solution coverage. On average, the probing based BFRs retain more than twice as many solutions as the BFRs produced without probing.

Table 1. Mean number of solutions retained for BFRs found with and without probing.

Problem (n, m, d, t)	No Probing	Probing	Ratio
(10, 10, 0.75, 0.3)	41.36	274.90	6.6
(10, 20, 0.5, 0.3)	774.26	3524.22	4.6
(10, 5, 0.25, 0.3)	121463.08	134494.04	1.1
(10, 5, 0.25, 0.7)	27.84	31.14	1.1
(10, 5, 0.5, 0.3)	587.42	2204.08	3.8
(10, 5, 0.5, 0.5)	4.10	4.28	1.0
(10, 5, 0.75, 0.3)	8.35	25.04	3.0

7.2. *Representing multiple 1-BFRs*

An orthogonal approach to finding 1-BFRs with higher solution coverage is to find multiple 1-BFRs and change the problem representation used online in order to support them. Any of the techniques used above for finding BFRs can be adapted to maintain more than one BFR: for example, we can adapt probing to keep the m BFRs with the highest solution retention.

Online, multiple BFRs can be incorporated into the original CSP by adapting the hidden variable method for transforming general CSPs into binary CSPs.[3] A single, new *BFR variable* is introduced to the problem with a domain corresponding to a set of labels, one for each BFR that is represented. The BFR variable is connected by a binary *BFR constraint* to each of the n original variables. Each constraint contains the valid tuples that map each BFR to the backtrack-free domain

of each original variable. For example, if BFR_i requires the domain of V_k to be $\{3, 4, 5\}$, the binary constraint between the BFR variable and V_k will contain the following tuples:[d] $(BFR_i, 3)$, $(BFR_i, 4)$ $(BFR_i, 5)$. Online, all variables are assigned in the original variable order. The BFR variable is not a decision variable, but AC is enforced on the BFR constraints. As we will prove below, such a representation is itself a BFR. In general, the hidden variable encoding requires the domain of the additional variable to be exponential in size as each label represents a satisfying tuple[3] or, in our case, a BFR. Provided that we only represent a constant number, m, of BFRs, this representation only adds a polynomial factor to the space complexity.

Example. To illustrate the multiple BFR idea, we return to our example problem and assume that we have two seeds: $(\langle V_1 = 6\rangle, \langle V_2 = 1\rangle, \langle V_3 = 3\rangle, \langle V_4 = 2\rangle)$ (as above) and $(\langle V_1 = 5\rangle, \langle V_2 = 2\rangle, \langle V_3 = 1\rangle, \langle V_4 = 3\rangle)$. The corresponding BFRs, found using arc consistency propagation as discussed in Section 4, are:

- BFR_1: $D_1 = \{6\}$, $D_2 = \{1\}$, $D_3 = \{2, 3, 4\}$, and $D_4 = \{2, 3, 4, 5, 7, 8, 9, 10\}$.
- BFR_2: $D_1 = \{5\}$, $D_2 = \{2\}$, $D_3 = \{1\}$, and $D_4 = \{3, 4, 6, 7, 8, 9, 10\}$.

Four new binary constraints are added to connect the new BFR variable, V_{BFR} with each of the other variables. The allowed tuples, in extensional form, are displayed in Table 2. The domains of the variables are: $D_1 = \{5, 6\}$, $D_2 = \{1, 2\}$, $D_3 = \{1, 2, 3, 4\}$, $D_4 = \{2, \ldots, 10\}$, and $D_{BFR} = \{BFR_1, BFR_2\}$. This multiple BFR representation retains all 28 solutions to the original problem.

Table 2. The n binary constraints added to represent multiple BFRs.

Constraint	Allowed Tuples
(V_{BFR}, V_1)	$(BFR_1, 6)$, $(BFR_2, 5)$
(V_{BFR}, V_2)	$(BFR_1, 1)$, $(BFR_2, 2)$
(V_{BFR}, V_3)	$(BFR_1, 2)$, $(BFR_1, 3)$, $(BFR_1, 4)$, $(BFR_2, 1)$
(V_{BFR}, V_4)	$(BFR_1, 2)$, $(BFR_1, 3)$, $(BFR_1, 4)$, $(BFR_1, 5)$, $(BFR_1, 7)$, $(BFR_1, 8)$, $(BFR_1, 9)$, $(BFR_1, 10)$ $(BFR_2, 3)$, $(BFR_2, 4)$, $(BFR_2, 6)$, $(BFR_2, 7)$, $(BFR_2, 8)$, $(BFR_2, 9)$, $(BFR_2, 10)$

When solving the problem online, the first decision will assign a value to variable V_1. Depending on which value is chosen (5 or 6), AC on the (V_{BFR}, V_1) constraint will assign V_{BFR} to BFR_2 or BFR_1 respectively. Subsequent AC propagation on the BFR constraints will reduce the domains to the corresponding backtrack-free domains.

[d] The constraint will, of course, also contain tuples representing the mapping between the other 1-BFR labels that are represented and the domain of V_k.

Theorem 1. The combination of backtrack-free representations using the adapted hidden variable model described above is itself a backtrack-free representation provided arc consistency is maintained on the BFR constraints.

Proof. We show that no assignment of an original variable to a value in its current domain can cause a domain wipe-out and therefore the representation is backtrack free.

Maintaining AC on the BFR constraints means that at any point in the search, each label remaining in the domain of the BFR variable has at least one support in the domain of each original variable and, conversely, that each value in the current domain of each original variable has at least one support in the domain of the BFR variable. It immediately follows that the assignment of any value in the current domain of any original variable cannot cause the BFR variable domain to be wiped out nor can subsequent AC propagation from the BFR domain variable cause the domain wipe-out of another original variable.

The only other opportunity for a domain wipe-out is if the assignment is inconsistent with all the values in the current domain of another original variable due to constraints in the original problem. This cannot happen because, as noted, each value in the domain of an original variable is present in at least one BFR represented by a corresponding label in the domain of the BFR variable. The definition of a BFR ensures that for any single variable assignment, there exists an assignment to each other original variable that is consistent with the original problem constraints. Therefore, no domain wipe-out can occur. □

Experiment. To investigate the feasibility of representing multiple BFRs, we found 10, 50 and 100 differing BFRs for problem instances ranging in tightness over the configuration $(15, 10, 0.7, t)$. The mean number of solutions retained are shown in Figure 6. For comparison, we plot the mean number of solutions in the original problem (all solutions), the number of solutions retained by the best BFR that could be found using probing (Iter1000rand), and the number of solutions in a single BFR using random pruning (Random). Representing multiple BFRs clearly increases the solution coverage over the other techniques. Note that we did not use probing to find the BFRs that were then made a part of the multiple BFRs. For the loosest problems, in fact, with probing we are able to find single BFRs that retain more solutions than a set of 10 BFRs. We believe that finding better individual BFRs and then combining them will provide an additional increase in the solution retention.

8. Discussion

In this section we discuss a number of additional points including extensions that we have not yet empirically investigated, the central issue that BFRs do not retain all solutions, the role of online consistency enforcement, and a broader perspective on a number of aspects of constraint research that is inspired by the BFR concept.

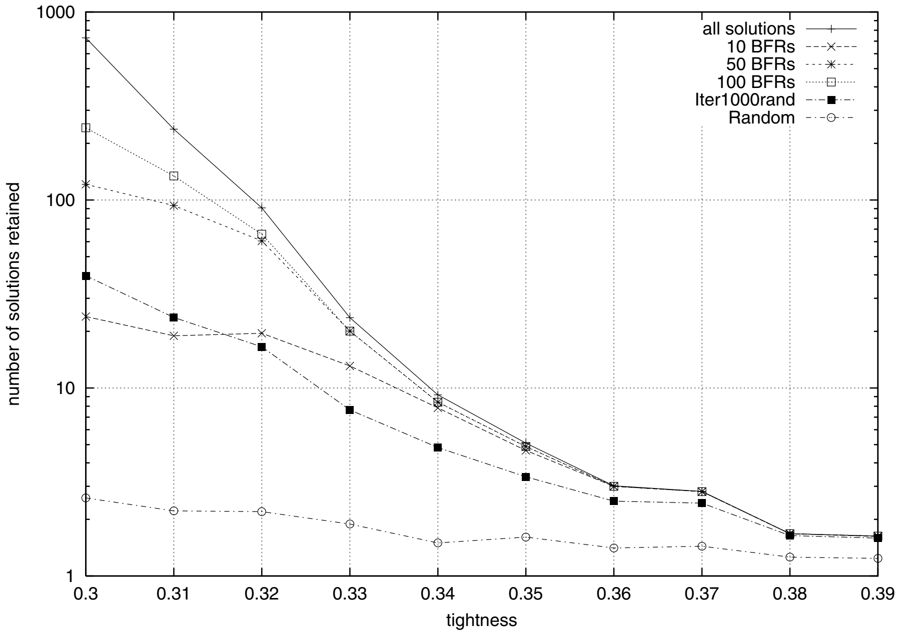

Fig. 6. Mean number of solutions retained by multiple BFRs representing 10, 50, and 100 BFRs, the mean number of solutions in the original problems, and the mean number of solutions represented in a single BFR found with probing.

8.1. *k-BFR and restricted BFR*

In our presentation of the BFR concept, algorithm, and experiments, we have focused on 1-BFR: parent solutions that do not extend to the child variable are removed by pruning a value in the non-extendible solution. Pruning values is equivalent to adding unary constraints but, in general, we do not have to be limited to unary constraints. Here we present two variations of this idea: k-BFR which limits the size of the constraint we can add to a problem and restricted BFR which limits us to tightening existing constraints.

8.1.1. *k-BFR*

Instead of pruning values from variable domains, we could add new constraints, up to arity k, as required to restrict the allowed tuples. 1-BFR is one end of a spectrum with the other end being full adaptive consistency. Between these extremes we can define a range of algorithms in which we can add constraints of arity up to k. For an n-variable problem, when k is 1 we have a value-removal algorithm, when k is $n - 1$ we have full adaptive consistency. As k increases the space complexity of our BFR increases but so does the solution retention.

For a fixed k, the space complexity of the algorithm remains polynomial in the problem size. Let n be the number of variables and d be the maximum domain size.

Let NG_k be the set of all possible no-goods of size k or less. The number of such no-goods, $|NG_k|$, is shown in Equation (1).

$$\begin{aligned}|NG_k| &= \binom{n}{k} \times d^k \\ &= \frac{1}{k!} \times \frac{n!}{(n-k)!} \times d^k \\ &= \frac{1}{k!} \times (n \times (n-1) \times \ldots (n-k+1)) \times d^k \\ &\leq O(n^k \times d^k)\,. \end{aligned} \tag{1}$$

The main benefit of 1-BFR is that we do not have to *add* the unary constraints because they are equivalent to removing domain values. Thus the space complexity does not increase. For k-ary constraints with $k > 1$, we have to actually add new constraints to the CSP and integrate them into the process of constraint solving. This addition yields extra space requirements which are exponential in general, though polynomial for a fixed k. However, by choosing an appropriate fixed value for k, the system designer can trade-off the available space with solution retention. With sufficient space, choosing $k = n - 1$ results in full adaptive consistency and therefore backtrack-free search without solution loss. With limited space, a fixed k can be chosen to use all space available, while losing some solutions. One interesting area of future work is to characterize the average solution loss for different k values. It may be the case, that significant solution coverage can be achieved with low k and therefore low space complexity.

Example. Returning to our example, from Section 4, here we demonstrate the application of 2-BFR while using the seed solution $(\langle V_1 = 6\rangle, \langle V_2 = 1\rangle, \langle V_3 = 3\rangle, \langle V_4 = 2\rangle)$ and arc consistency propagation.[e] With 2-BFR, we add binary no-goods specifying pairs of values that are disallowed. As we use a seed solution, we must ensure that each no-good posted does not appear in the seed. The offline, 2-BFR processing is as follows:

1. We run arc consistency on the initial problem resulting in the following domains: $D_1 = \{3, 4, 5, 6\}$, $DV_2 = \{1, 2, 3, 4\}$, $D_3 = \{1, 2, 3, 4\}$, and $D_4 = \{1, \ldots, 10\}$.
2. In the first iteration, two parent solutions (as above) cannot be extended: $(\langle V_1 = 5\rangle, \langle V_2 = 2\rangle, \langle V_3 = 2\rangle)$ and $(\langle V_1 = 6\rangle, \langle V_2 = 1\rangle, \langle V_3 = 1\rangle)$. We add two no-goods $\neg(\langle V_2 = 2\rangle, \langle V_3 = 2\rangle)$ and $\neg(\langle V_2 = 1\rangle, \langle V_3 = 1\rangle)$ to remove both non-extendible parent solutions.[f] Arc consistency propagation cannot remove any values at this stage.

[e]Given that $n = 4$, 3-BFR is equivalent to full adaptive consistency and so would retain all solutions.

[f]Had we added $\neg(\langle V_1 = 5\rangle, \langle V_2 = 2\rangle)$, the BFR would end up being identical to the 1-BFR found above. We chose to demonstrate the possible added power of 2-BFR.

3. In the next iteration, the algorithm examines the parent subproblem of V_3 which is: $(\{V_1, V_2\}, \{D_1, D_2\}, V_1 + V_2 = 7)$ with $D_1 = \{3, 4, 5, 6\}$ and $D_2 = \{1, 2, 3, 4\}$. It has the solutions $\{(\langle V_1 = 3\rangle, \langle V_2 = 4\rangle), (\langle V_1 = 4\rangle, \langle V_2 = 3\rangle), (\langle V_1 = 5\rangle, \langle V_2 = 2\rangle), (\langle V_1 = 6\rangle, \langle V_2 = 1\rangle)\}$. The final two solutions extend so we need to add binary no-goods to remove the first two solutions. We add $\neg(\langle V_1 = 3\rangle, \langle V_2 = 4\rangle)$ and $\neg(\langle V_1 = 4\rangle, \langle V_2 = 3\rangle)$. As these value are sole supports (i.e., $\langle V_1 = 3\rangle$ is the only supporting value for $\langle V_2 = 4\rangle$ and vice versa), arc consistency removes both pairs of values from the domains.
4. Finally, the parent subproblem of V_2 is processed. Both values of V_1 are consistent with one value of V_2. No pruning is necessary and the algorithm terminates with the following BFR: $D_1 = \{5, 6\}$, $D_2 = \{1, 2\}$, $D_3 = \{1, 2, 3, 4\}$, and $D_4 = \{1, \ldots, 10\}$. This BFR covers all 28 solutions to the original problem. The no-goods must be retained and added to the online representation.

8.1.2. *Restricted BFR*

If the constraints are represented extensionally, instead of adding new constraints to the problem model, we can specify that the changes that can be made to remove a non-extendible parent subproblem must be limited to tightening existing constraints. We call this variation *restricted BFR*. For example, assume that the partial assignment, $\langle X_1 = v_1\rangle, \ldots, \langle X_m = v_m\rangle$, does not extend to a solution and a constraint c over $X_1, \ldots, X_m$ exists. Restricted BFR simply removes the tuple $(v_1, \ldots, v_m)$ as a satisfying tuple of c. If all such constraints exist, we can achieve adaptive consistency without any extra memory consumption. Constraint c may not exist and therefore the algorithm has to consider constraints over subsets of the variables $\{X_1, \ldots, X_m\}$. Given binary constraints between some pairs of the variables, we can remove the tuple (v_i, v_j) from any constraint where X_i and X_j are involved in the dead-end. A reasonable approach is to identify the highest arity constraint involved in a dead-end and remove a single tuple. To ensure correctness, we need to view the domain of each variable as an extensionally represented constraint. In the extreme, a parent subproblem may have no non-unary constraints and therefore we are forced to remove any non-extendible solutions by removing a unary tuple from a parent variable (i.e., remove a value from the variable's domain as in 1-BFR).

k-BFR and restricted BFR suggest a number of research questions. What is the increase in solution coverage and quality that can be achieved without extra space consumption? If we allow new constraints to be added, how do we choose an appropriate k value? How does a good k relate to the arity of the constraints in the original CSP? Given the spectrum that exists between 1-BFR and adaptive consistency, we believe that future empirical work on these and related questions can be of significant use in making adaptive consistency-like techniques more practically applicable.

8.2. *Coming to terms with solution loss*

We have shown that, through various implementations and extensions to the basic BFR concept, we can find problem representations that guarantee backtrack-free search while preserving many of the solutions, including an optimal solution, to the original problem. Despite these results, the central challenge to the BFR concept is that solutions are lost: perfectly good solutions to the original CSP are not represented in its BFR. A problem transformation that loses solutions is a radical, perhaps heretical, concept. Solution loss may be too high a price to pay to remove the need to undo previous decisions.

There are three characteristics of problem representations that are relevant here: space complexity, potential number of backtracks, and solution loss. It is unlikely, theoretically, that we can simultaneously achieve polynomial space complexity, a guarantee of backtrack-free search, and zero solution loss. Therefore, we need to reason about the trade-offs among these characteristics. Precisely how we make these trade-offs depends on the application requirements and the BFR concept allows us to build systems that reason about these trade-offs in at least two different ways: either as part of the offline processing or as a combination of the offline and online processing.

8.2.1. *Making the trade-off offline*

By combining k-BFR and restricted BFR as well as choosing the k value, we can create an offline algorithm that produces a representation that directly trades-off solution loss, storage space, and backtracks. An extension to Seeded-1-BFR (Algorithm 1), could change line 7 from the selection of a value to prune to a three-way decision. Based on problem instance measurements such as the number of values already pruned, estimates of the remaining solutions, and dead-ends, and other characteristics the decision at line 7 would be to:

- Remove a value (1-BFR) or a valid tuple from an existing constraint (restricted BFR), thereby maintaining the current storage requirements and removing the backtrack, while potentially losing solutions; or
- Add a no-good (k-BFR or directional arc-consistency), removing a backtrack and not introducing any solution loss while increasing the storage requirements; or
- Do nothing, maintaining all current solutions and current storage requirements while leaving a potential backtrack to be found online.

For example, one policy would be to add no-goods while space is available and then transition to restricted BFR when an appropriate constraint exists and, to making no change to the problem otherwise.

Such an approach allows the offline algorithm to reason in an agent-like manner to explicitly take into account the trade-off among storage space (how much is available?), online backtracks (can "shallow" backtracks be recognized and allowed since the online effort to escape them will be small?), and solution loss. It may even be

possible to quantify the trade-offs and generate an optimal problem representation: online storage costs $\$x$ per MB, each solution available results in an $y\%$ increase in revenue through higher sales but each backtrack encountered results in a loss of potential revenue of $z\%$ due to abandoned transactions. With such numbers, a representation could be created that would maximize the expected revenue.

8.2.2. *Making the trade-off both offline and online*

Alternatively, the system designer and user can collaboratively make the three-way trade-off in two steps. First, the system designer can decide on the storage requirements by choosing the arity, k, in a k-BFR approach and the number of BFRs to be represented. In the extreme, a single $(n-1)$-BFR achieves full adaptive consistency and so, if the memory space is available, a zero backtrack, zero solution loss BFR can be achieved. The system designer, therefore, makes the decision about the trade-off between the number of solutions that can be found without backtracking and the space complexity. The use of seed solutions means that each BFR can be built around solutions preferred by the system designer: guaranteeing a minimal set of desirable solutions. Second, multiple BFRs together with the original CSP can be used online to allow the user to make the trade-off between solution loss and backtracks. The BFRs create a partition of the domain of each variable: those values that are guaranteed to lead to a solution without backtracking and those for which no such guarantee is known. These partitions can be presented to the user by identifying the set of "safe" and "risky" options for a particular decision. If the user chooses to make safe decisions, the BFRs guarantee the existence of a solution. If the user decides to make a risky decision, the system can (transparently) transition to standard CSP techniques without solution guarantees. This allows the user to make the trade-off between backtracks and solution loss: if the user has definite ideas about the desired configuration, more effort in terms of undoing previous decisions may be necessary. If the user prefers less effort and has weaker preferences, a solution existing in one of the BFRs can be found. In this scenario, all solutions to the original problem are available to the user. The decisions of the system designer concern the memory usage and the relative number of "safe" options that the user will be faced with: a higher memory usage via higher k value for k-BFR or more solutions in the multiple BFR will increase the number of solutions that are accessible by a series of safe values. However, the user can still find any solution provided he or she is willing to risk backtracking.

8.3. *Online consistency enforcement*

Aside from the immediately preceding discussion of safe and risky variables and the multiple BFR representation, we have not addressed the issue of the algorithm that will be used online to find solutions. We have, in fact, assumed a simple backtracking algorithm, without any consistency enforcement, will be used. If we make

stronger assumptions about the online processing, we can create better BFRs. Intuitively, if we use MAC online, we will encounter fewer dead-ends and therefore our offline processing has to remove fewer dead-ends. For example, recall the small graph coloring problem presented in Section 1: we have variables $\{X, Y, Z\}$, colors $\{red, blue\}$, constraints (X, Z) and (Y, Z), and a lexicographic variable ordering. Above, we created a BFR by removing the value red from both X and Y. If we assume that we are running MAC online, no pruning is required. X is the first variable to be assigned and, regardless of which value is chosen, arc consistency propagation will immediately assign the other two variables to consistent values. Interestingly, this example illustrates that we can weaken the notion of a backtrack-free representation to make it *dependent on the online algorithm.* A BFR built for MAC will not be a BFR for simple backtracking; though the converse is true.

More precisely, suppose in the static value ordering we have a parent problem, PP_k, that contains at least two variables, V_i, V_j that do not share a constraint. Further suppose that there is a path, P, of constraints $(V_i, V_x), ...(V_y, V_j)$ consisting of variables later in the variable ordering than both V_i and V_j. Finally, assume that the tuple $t = (\langle V_i = a \rangle, \langle V_j = b \rangle)$ is arc inconsistent based given the constraints in P. An online algorithm using simple backtracking may assign the values in tuple t. At some point such an assignment will result in a domain wipe-out because it is globally inconsistent. Therefore, any BFR to be used by a simple backtracking algorithm must remove one of the assignments in t even though they may both, individually, participate in other solutions. In contrast, if the online algorithm is MAC, both assignments in t can remain because MAC will ensure that they are not both assigned in the same search path.

A BFR algorithm can be easily modified to ensure that only those dead-ends that exist for a specific online algorithm will be pruned. The crux of the change for Seeded-1-BFR (Algorithm 1) is the enumeration of the parent solutions at line 6. Rather than a simple enumeration, we search for parent solutions by assigning values to variables and enforcing the target level of consistency on the entire problem after each assignment. Such enforcement may, as in the graph coloring example, remove values from the other parent variables, that would otherwise lead to a non-extendible parent solution. When we achieve a full assignment of the parent variables that does not extend, we know that this assignment may be encountered online and therefore needs to be removed.

9. Conclusion

In this paper we identify, for the first time, the three-way trade-off between space complexity, backtrack-free search, and solution loss. We present an approach to obtaining a backtrack-free CSP representation that does not require additional space and we investigate a number of variations on the basic algorithm including the use of seed solutions, arc consistency, and a variety of pruning heuristics. We

have evaluated experimentally the cost of obtaining a BFR and the solution loss for different problem parameters. Overall, our results indicate that a significant proportion of the solutions to the original problem can be retained especially when an optimization algorithm that specifically searches for such "good" BFRs is used. We have seen how multiple BFRs can cover more of the solution space. Furthermore, we have argued that BFRs are an approach that allows reasoning about the trade-off between the space complexity of the problem representation, the backtracks that might be necessary to find a solution, and the loss of solutions.

We view this paper as an initial investigation of the implications surrounding questioning a single assumption: in an offline/online setting, it is always necessary to retain all solutions to a CSP. As noted, this is a radical idea and, as such, may seem too bizarre to merit further thought. We have attempted, in this paper, to argue against this position: though radical, the idea of preprocessing CSPs to achieve backtrack-free search at the cost of the removal of some solutions, generates a number of research questions and opportunities for future work. We hope to investigate these opportunities together with other members of the research community.

Acknowledgments

This material is based upon works supported by the Science Foundation Ireland under Grant No. 00/PI.1/C075 and No. 05/IN/1886, the Embark Initiative of the Irish Research Council of Science Engineering and Technology under Grant PD2002/21, the Natural Sciences and Engineering Research Council of Canada, and ILOG, S.A. In addition to the sponsors of this research, the authors would like to thank Nic Wilson, Ken Brown, and Gérard Verfaillie for useful discussions and comments.

Appendix A. General Formulation and Proofs

The correctness of the Seeded-1-BFR algorithm presented above (Algorithm 1) follows from the well-established correctness of methods for finding a single solution to a constraint satisfaction problem. However, in general, creating a 1-BFR by removing values from domains does not require a seed solution nor consistency enforcement. Therefore, in this appendix, we present a general algorithm for 1-BFR and prove its correctness.

Algorithm 2 is an algorithm to find a BFR (if one exists) for a CSP. The primary change from Seeded-1-BFR is that without a seed solution, it is possible that the pruning that is done to remove non-extendible parent subproblems may lead to a variable with an empty domain. This change requires the ability to "backtrack" through the pruning decisions to make alternate choices which in turns results in substantially different pseudocode. There are two recursive calls: one at line 9 if all the parent solutions extend and one at line 20 after each non-extendible parent solution has been removed.

Algorithm 2 General-1-BFR(k): Obtains a BFR

Require: Initialize $S^k_{done} = \emptyset$ for all $k = 1..n$
1: **if** $k = 1$ **then**
2: return Success
3: **end if**
4: **if** solutionsOf(PP_k) $= \emptyset$ **then**
5: return Failure
6: **end if**
7: $D := \{S \in solutions(PP_k)$ s.t. $doesNotExtend(S, V_k) \wedge S \notin S^k_{done}\ \}$
8: **if** $D = \emptyset$ **then**
9: return General-1-BFR($k - 1$)
10: **else**
11: choose $S \in D$
12: **end if**
13: $r :=$ Failure
14: **while** $S \neq \emptyset$ AND $r =$ Failure **do**
15: choose $a \in S, a = (V_i, X_j)$
16: $S := S \setminus \{a\}$
17: $D_i := D_i \setminus \{X_j\}$
18: **if** $D_i \neq \emptyset$ **then**
19: $S^k_{done} := S^k_{done} \cup \{S\}$
20: $r :=$ General-1-BFR(k)
21: **end if**
22: **if** r = Failure **then**
23: $S^k_{done} := S^k_{done} \setminus \{S\}$
24: $D_i := D_i \cup \{X_j\}$
25: **end if**
26: **end while**
27: return r

The correctness of the above algorithm is not as straightforward as with Seeded-1-BFR. It is clear that a BFR to a soluble problem must exist; any individual solution is a BFR. However, it is not immediately obvious that General-1-BFR will correctly return Failure for an insoluble problem or that, in removing values from a soluble problem, it might not remove all solutions.

Theorem 2. If P is soluble, General-1-BFR will find a backtrack-free representation.

Proof. Proof by induction.

Inductive Step. If we have a solution S to PP_k we can extend it to a solution to P_k without backtracking. Solution S restricted to the parents of V_k is a solution

to the parent subproblem of V_k. There is a value, b, for V_k consistent with this solution, or else this solution would have been eliminated by General-1-BFR. Since we only need worry about the consistency of b with the parents of V_k, adding b to S gives us a solution to P_k.

Base Case. PP_1 is soluble, i.e. the domain of V_1 is not empty after General-1-BFR(1). Since P is soluble, let S be one solution such that $(V_1, s_1) \in S$. We will show that if it does not succeed otherwise, General-1-BFR will succeed by providing a representation that includes s_1 in the domain of V_1. We will do this by demonstrating, again using induction, that in removing a solution to a subproblem, PP_k, General-1-BFR will always have a choice that does not involve a value of S. Suppose General-1-BFR has proceeded up to V_k without deleting any value in S. It is processing V_k and a solution S_j to the parent subproblem does not extend to V_k. If all the values in S_j are in S, then there is a value in V_k that is consistent with them, namely the value for V_k in S. So one of the values in S_j must not be in S, and General-1-BFR can choose at some point to remove it. (The base step for V_n is trivial.) Since General-1-BFR tries, if necessary, all choices for removing values (the *while*-loop in Algorithm 2), it will eventually choose, if necessary, not to remove any component in S, including (V_1, s_1). □

Theorem 3. If P is insoluble, General-1-BFR will return Failure.

Proof. Proof by induction. We will show that if P_k is insoluble, then after General-1-BFR processes it, P_{k-1} is insoluble. Thus eventually General-1-BFR will backtrack when P_1 becomes insoluble (the domain of V_1 is empty) if not before, and will eventually run out of choices to try, and return Failure.

Inductive Step. Suppose P_k is insoluble. We will show that P_{k-1} is insoluble after General-1-BFR has processed P_k in a proof by contradiction. Suppose S is a solution of P_{k-1}. Then S restricted to the parents of V_k, S_k, is a solution of the parent subproblem of V_k, which is a subproblem of P_{k-1}. There is a value b of V_k consistent with S_k, for otherwise S_k would have been eliminated during processing of P_k. But if b is consistent with S_k, S_k plus b is a solution to P_k. Contradiction.

Base Case. $P_n = P$ is given insoluble. □

References

1. J. Amilhastre, H. Fargier, and P. Marquis, Consistency restoration and explanations in dynamic csps – application to configuration. *Artificial Intelligence*, 135:199–234, 2002.
2. J. C. Beck, P. Prosser, and R. J. Wallace, Trying again to fail first. In B. Faltings, A. Petcu, F. Fages, and F. Rossi, editors, *Recent Advances in Constraints*, volume 3419 of *Lecture Notes in Artificial Intelligence*, pages 41–55. Springer-Verlag, 2005.

3. R. Dechter, On the expressiveness of networks with hidden variables. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI90)*, pages 556–562, 1990.
4. R. Dechter, *Constraint processing*. Morgan Kaufmann Publishers, 2003.
5. R. Dechter and J. Pearl, Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1987.
6. E. C. Freuder, Synthesizing constraint expressions. *Communications of the Association for Computing Machinery*, 21(11):958–966, November 1978.
7. E. C. Freuder, R. J. Wallace, and R. Heffernan, Ordinal constraint satisfaction. In *5th International Workshop on Soft Constraints-Soft 2003*, 2003.
8. E.C. Freuder, A sufficient condition for backtrack-free search. *Journal of ACM*, 29(1):24–32, 1982.
9. E.C. Freuder, Complexity of k-tree structured constraint-satisfaction problems. In *Proceedings of the Eigth National Conference on Artificial Intelligence (AAAI-90)*, pages 4–9, 1990.
10. P.A. Geelen, Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI-92)*, pages 31–35, 1992.
11. P.D. Hubbe and E.C. Freuder, An efficient cross-product representation of the constraint satisfaction problem search space. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 421–427, 1992.
12. D. Lesaint, Maximal sets of solutions for constraint satisfaction problems. In *Proceedings of the Eleventh European Conference on Artificial Intelligence*, pages 110–114, 1994.
13. O. Lhomme, Consistency techniques for numeric CSPs. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, volume 1, pages 232–238, 1993.
14. N. R. Vempaty, Solving constraint satisfaction problems using finite satate automata. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 453–457, 1992.
15. R.J. Wallace, Factor analytic studies of CSP heuristics. In *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP-05)*, 2005. Available at: http://4c.ucc.ie/web/people.jsp?id=34.
16. J.-P. Watson, J. C. Beck, A. E. Howe, and L. D. Whitley, Problem difficulty for tabu search in job-shop scheduling. *Artificial Intelligence*, 143(2):189–217, 2003.
17. R. Weigel and B. Faltings, Compiling constraint satisfaction problems. *Artificial Intelligence*, 115(2):257–287, 1999.