Preprocessing Expression-Based Constraint Satisfaction Problems for Stochastic Local Search

Sivan Sabato and Yehuda Naveh

IBM Haifa Research Lab, Haifa University Campus, Haifa 31905, Israel
{sivans,naveh}@il.ibm.com

Abstract. This work presents methods for processing a constraint satisfaction problem (CSP) formulated by an expression-based language, before the CSP is presented to a stochastic local search solver. The architecture we use to implement the methods allows the extension of the expression language by user-defined operators, while still benefiting from the processing methods. Results from various domains, including industrial processor verification problems, show the strength of the methods. As one of our test cases, we introduce the concept of random-expression CSPs as a new form of random CSPs. We believe this form emulates many real-world CSPs more closely than other forms of random CSPs. We also observe a satisfiability phase transition in this type of problem ensemble.

1 Introduction

The most important aspect of constraint programming (CP) over other variable-assignment paradigms (e.g., Satisfiability or integer linear programming) is its ease of modeling. CP allows users to describe the problem at hand in a way that is close to the problem's domain, as opposed to a formal language derived from the solution scheme in other methods. One generic way to allow this natural modeling is to define an expression-based language with simple operators that have a wide range of semantics. Operators may be arithmetic (e.g., +,-), logical (e.g., `and`, `or`), set operators (e.g., `member-of`), or problem-domain specific. They may be binary (e.g., `equal-to`, `less-than`) or global (e.g., `all-different`, `equal-sum`). One example of a generic expression language is OPL [1], supported by ILOG.

Some classes of constraint satisfaction problems (CSPs) are recognized as not easily solved by systematic methods, and stochastic local search (SLS) methods need to be called upon [2]. In this paper we show that the ease with which SLS methods can solve a CSP model written in an expression language highly depends on the way the model is processed and analyzed before it is presented to the SLS solver. We present generic processing algorithms that transform the input CSP model into an SLS model that is easier to solve in many cases. As in [3], the processing algorithms use interfaces (or abstract methods) to support the addition of any user-defined operator. However, while previous works focus on abstracting methods related to the search phase [4, 3], we are interested in processing the CSP model itself, *before* entering the search phase.

One of the main tasks in developing an SLS solver involves enhancing its ability to escape local minima in the topography defined by the cost function of all complete assignments. Therefore, SLS solvers (e.g., Walksat [5] or COMET [3]), can incorporate many types of meta-heuristics, such as simulated annealing [6], min-conflicts [7], Tabu Search [8], or variable-neighborhood search [9], designed to escape local minima. Still, for all these methods, it is highly beneficial to have the CSP mapped into a topography with fewer local minima and plateaus.

The abstract interfaces and concrete methods we present aim to achieve this goal in a generic way. More specifically, real world CSPs are sometimes composed of differently structured constraints, written independently from each other. The resulting problem of 'bad models' is known [10], and processing algorithms aimed at improving the models exist for non-SLS search schemes (e.g., in Satisfiability [10] and arc-consistency methods [11]). Here we extend these methods to SLS.

This paper is outlined as follows. Section 2 defines an expression language we use to model the input CSP. Section 3 describes the architecture of the SLS solver and its use of operators in the expression language. Section 4, which forms the main part of the paper, refines this architecture to support the processing methods and presents the processing algorithms. Experimental results are shown in Section 5, where the concept of random expression CSP is also presented.

2 Expression-Based CSPs

The formal grammar of the example language we use as input throughout the paper is listed in Figure 1. This language captures many of the basic constructs expected from a general-purpose expression-based CSP language. While additional generic or domain-specific operators may be added, this language is powerful enough to easily model many of the problems in CSPLib (`http://www.csplib.org/`), as well as a large number of real world problems.

In Figure 1, words in upper case stand for non-terminals and underlined words are reserved. The entire CSP is generated from the non-terminal P. A CSP description is comprised of a list of declarations of integer variables and their domains, and a list of constraints created from logical, arithmetic, and other operators. Domains of variables are defined as ranges of integers. An example for the use of the input language is given in Figure 2. We will sometimes formulate CSP descriptions more loosely for ease of presentation, when it is obvious how one would translate them to a valid description using our grammar.

3 System Architecture

3.1 Internal CSP Representation

An expression-based CSP is represented by a rooted tree as exemplified in Figure 3. The leaves of the tree are mapped to variables or constants and the nodes to

```
P ⟶ VARDECL constraints CONSDECL
VARDECL ⟶ VD ; VARDECL | ϵ
CONSDECL ⟶ CONS; CONSDECL | ϵ
VD ⟶ VAR RANGES
RANGES ⟶ RANGES, [NUM, NUM]
RANGES ⟶ [NUM, NUM]
CONS ⟶ (CONS) OP (CONS)
CONS ⟶ not (CONS)
CONS ⟶ ((EXP) COMPARE (EXP))
CONS ⟶ all-diff( VARLIST )
CONS ⟶ some-equal( VARLIST )
EXP ⟶ ((EXP) EXP_OP (EXP))
EXP ⟶ NUM
EXP ⟶ VAR
VARLIST ⟶ VAR, VARLIST | ϵ
OP ⟶ and | or | implies | iff
COMPARE ⟶ = | ≠ | ≥ | ≤ | < | >
EXP_OP ⟶ + | − | × | /
NUM ⟶ [0-9]+
VAR ⟶ [A-Za-z_][A-Za-z0-9_]+
```

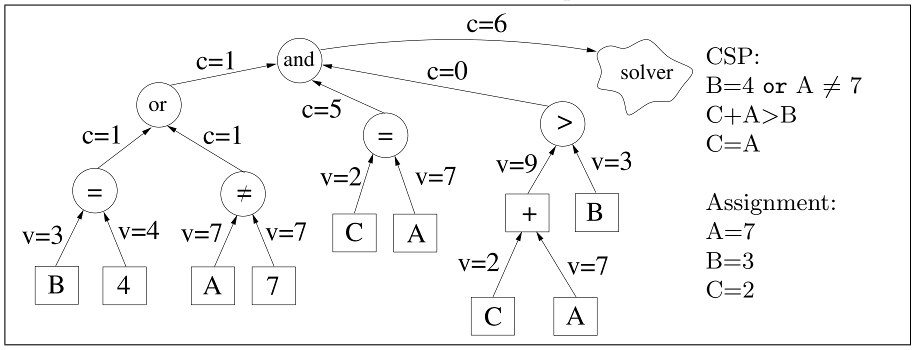
Fig. 1. Input language grammar

```
Bread [0,2], [4,5];
Milk [0,2];
Cheese [0,2];
Butter [0,2];
Apple [0,3], [6,6];
Payment [0,1000];
constraints
(Bread = Cheese + Butter);
(Apple < 2) implies (Bread > 1);
(Cheese + Butter < 3);
((Apple = 1) or (Bread = 0));
(all-diff(Milk, Cheese, Butter));
(Payment = Milk×3 + Cheese×4);
```

Fig. 2. Input language example

operators. Nodes that are mapped to arithmetic operators represent *expressions*, while nodes that are mapped to comparison operators or to logical operators represent *sub-constraints*. The root node is mapped to an *and* operator and its children are the individual constraints in the input CSP.

Fig. 3. A CSP tree and the flow of information when calculating an assignment's cost, where 'c' stands for constraint cost (or penalty) and 'v' stands for expression value. Natural cost functions were chosen for the various operators.



A sub-constraint node is defined by its operator and by its children. We refer to sub-constraints with comparison operators as *atomic*, since their children are expressions and not other sub-constraints. We refer to sub-constraints with logical operators as *compound*. Global constraints such as `all-different` can be implemented either as atomic operators or as compound expressions.

Nodes are implemented as objects that are sub-classed from an abstract sub-constraint object or from an abstract expression object. Sub-constraint objects implement (among other) the usual interface:

`CalculateCost(Assignment)`

which returns the cost that the sub-constraint assigns to `Assignment`. Expression objects implement the interface:

`CalculateValue(Assignment)`

This interface returns the value of the expression in `Assignment`. The solver only has access to the interfaces of the root of the constraint tree. The computational cost of calculating the cost of the CSP tree for any single assignment is linear in the number of nodes in the tree.[1]

The cost function implemented for any sub-constraint is zero if and only if the sub-constraint is satisfied; otherwise it is positive. In addition, this cost function should exhibit the best possible fitness-distance correlation (FDC) [12]: The further the state is from a solution, the higher the cost of the state should be. There are no conceptual problems with the definition of a cost function for atomic sub-constraints, whether simple expressions or global constraints [13]. However, implementing the cost of a compound sub-constraint is not as straightforward, since it should depend only on the cost of its child sub-constraints. This is because the same sub-constraint (and hence the same implementation of its cost function) may span very different sub-trees of sub-constraints and variables. Some operators may benefit from the fact that there exists a cost function whose topography is related to the topographies of the child constraints. This is the case for `and` and `or` operators with the natural costs of `sum` and `min` of children, respectively. However, for other operators, it may be impossible to implement a cost function that reflects the topographies of the child constraints. Notable examples are `not`, `implies` and `iff`. We address this problem in detail in Section 4.1.

3.2 The Search Scheme

A typical scheme of a greedy SLS algorithm is outlined in Algorithm 1. The algorithm starts with an initial assignment. In each iteration, it generates a set of steps from the current assignment to a set of new assignments, and calculates the cost of each of the resulting assignments. If at least one step results in an assignment with a lower cost, this assignment becomes the current one. Otherwise, the topography of the problem is modified by giving a larger weight to constraints that are not satisfied by the current assignment. The algorithm stops when a zero-cost assignment is reached or at timeout. One of the main differentiators between solvers that use this scheme is the neighborhood function that generates S.

[1] Powerful heuristics that exploit the fact that each step in the search space usually changes the cost of only a few of the CSP tree nodes are also applied, but are similar to those reported elsewhere [3].

Algorithm 1 General Scheme of Search Algorithm

```
Initialize $A$ to an initial complete assignment
repeat
  repeat
    if $cost(A) = 0$ then
      Return
    end if
    Initialize $S$ to a set of possible steps
    Calculate $cost(A + s)$ for all $s \in S$
    $s_1 \leftarrow \text{argmin}_{s \in S}\, cost(A + s)$
    if $cost(A + s_1) < cost(A)$ then
      $A \leftarrow A + s_1$
    end if
  until $cost(A + s_1) \geq cost(A)$
  Set constraint weights such that unsatisfied constraints get a larger weight
until Timeout is reached
```

3.3 The Search Space

In our implementation, we follow Algorithm 1 and define the neighborhood of an assignment to be the assignments in which up to M bits are changed in the 2's complement bit-representation of the integer variables of the CSP, where M is given, and calculated by dynamic heuristics [14]. In this representation, an additional unary constraint is added for each variable whose domain is not a power of two, to enforce the domain requirement. This simple approach is favorable for some constraint types (e.g., `less-than` and `greater-than`). It is also particularly useful in hardware verification where many constraints are defined on bit-ranges [15]. However, most methods presented below can be generalized to other representation schemes and more sophisticated neighborhood functions.

4 Model Processing for SLS

In this section we present several methods for processing the input CSP model. The processing methods rely on implementing specific interfaces for sub-constraints and expressions. Unlike the interfaces `CalculateCost()` and `CalculateValue()` that define the *semantics* of the objects, the interfaces presented below are used only by the processing algorithms and do not change the semantics of the CSP. Hence, it is not necessary to implement all interfaces for all operators in the language.

We demonstrate our methods on the grammar of Figure 1, but the methods can be applied to grammars that use other operators by implementing the required abstract interfaces for each operator. The modeler of a CSP may thus experiment with different types of newly-defined operators, without changing the processing or search algorithms. This extends the regular generic interface of the search phase to the pre-search phase. To demonstrate the operation of the processing methods, we present the following simple CSP example.

```
V1,V2,V3,V4,V5,V6 [0,5000];
constraints
1. (((V3 ≠ V5+3) or (V5 > 10)) implies (V2 ≠ V3));
2. ((V4 > 11) and (V2 = V3));
3. ((V1 < 5) or ((V1 < 12) and (V1 > V3−4)));
```

This problem exemplifies a mixture of logical and arithmetic operators and a diverse structure of constraints often found in real world problems. We define the variable domains in this example to be relatively large, since the problem is small for didactic reasons and we want to keep the search space large. In the 2's complement representation, each variable is represented by 13 bits and constraints of the form $V_n \leq 5000$ are added.

4.1 Transformation to Negation Normal Form

In Section 3.1, we mentioned `sum` and `min` as reasonable cost functions for the Boolean operators `and` and `or`. We now show that other Boolean operators may present an inherent problem to the tree structure of the cost function.

Consider the unary operator `not`. Let us look at a sub-constraint node $C = not(C')$. We need to implement a cost function f_{not} such that on any complete assignment A, $cost(C, A) = f_{not}(cost(C', A))$. For f_{not} to be a legal cost function, it must output zero if C' is not satisfied by A (i.e., if $cost(C', A) > 0$) and non-zero if C is satisfied (i.e., if $cost(C', A) = 0$). The only functions that obey these limitations are of the following form, for some $k > 0$:

$$f_{not}(c) = \begin{cases} k & \text{for} \quad c = 0 \\ 0 & \text{otherwise} \end{cases}$$

This implies that f_{not} has zero gradient when it is unsatisfied, leaving no possibility of finding a satisfying assignment using gradient descent ('greedy') methods. In other words, the `not` operator 'hides' information on the location of minima in its child sub-constraint cost function. A similar problem is encountered with the logical operators `implies` and `iff`. All these operators are, however, a basic part of any natural expression language. Our first processing method therefore transforms the model to negation normal form (NNF), which substitutes the ill-behaved operators with the better-behaved `and` and `or`.

The *NNF transformation* is applied in the regular manner to compound sub-constraints realizing the above operators. Atomic sub-constraint operators need to implement the following interface in order to take advantage of this method:

`GetNegatedOperator()`.

For example, the implementation of the = operator would return the operator ≠, the operator > would return the operator ≤, and the global operator `all-different` would return its negation `some-equal`. The transformation is applied to the input CSP recursively from top to bottom. Its time-complexity is linear in the size of the CSP tree and the resulting tree is about the same size as the original one. In our example CSP, the NNF transformation changes constraint No. 1 to: (((V3 = V5 + 3) `and` (V5 ≤ 10)) `or` (V2 ≠ V3));

4.2 Reducing the Search Space Size

Two processing methods presented here perform low-cost inferences that enable the pruning of large parts of the search space for which search is useless. These inferences are special and simple cases of domain reductions that could have also been achieved using propagators in an arc-consistency algorithm. While there are many ways to combine arc-consistency with local search (see [16] for an early example), the overall search may be prohibitive in problems that are not suitable for arc-consistency methods. In contrast, here we limit our processing methods to ones whose processing cost is linear in the size of the CSP tree, and we apply the methods only on the initial CSP model before starting SLS. Hence, our search is dominated by SLS and the inference cost is usually negligible.

The two methods presented in this section rely on the dimensions defining the search space. In a bit-representation (sub-section 3.3), each dimension is defined by a single bit. In other representations, each dimension may correspond to a single CSP variable or to any combination of variables' values.

If the cost function does not depend on the dimension's value in any assignment, the solver does not need to change this value during search. We term such a dimension *unimportant*. For example, in the bit-representation, the least-significant-bit of a variable X in the constraint "X > 5" is unimportant. Alternatively, if we can infer in advance that the value in a given dimension is the same for all solutions, the solver can set the value to this fixed value and remove the dimension from the search space. We refer to such a dimension as *predetermined*. An example of a predetermined dimension in the bit-representation is the least-significant-bit of a variable V constrained by "V `mod` 2 = 0".

Finding Unimportant Dimensions We find unimportant dimensions by having sub-constraint- and expression-objects implement the interface:

`GetDependentDimensions()`

which returns the list of dimensions that may affect the object's cost or value. Dimensions that do not appear in the list returned by the root node are unimportant. Note that finding all unimportant dimensions in a general CSP is NP-hard: If the CSP includes one constraint that is a 3-CNF formula, deciding whether there are any important dimensions is tantamount to finding whether the formula is satisfiable.

In our CSP example, the variable V6 is not used by any constraint; Hence, all its bits are found to be unimportant. The same applies to the two least-significant-bits of V4. Additional unimportant bits of this CSP will be found after other processing methods are applied.

Finding Predetermined Dimensions We find predetermined dimensions (PDs) using a recursive and iterative algorithm: Each sub-constraint node implements the interface

`InferPredeterminedDimensions(CurrentPredeterminedDimensions)` which returns a set of PDs along with their predetermined value. For atomic sub-constraints, the interface uses the currently known PDs and tries to find new PDs according to its own semantics. For example, in a bit-representation, in the atomic constraint "X $<$ 5", the bits higher than the 3 least-significant-bits in X are zero.

For a compound sub-constraint, the interface calls `InferPredetermined-Dimensions(CurrentPredeterminedDimensions)` for each of its child sub-constraints, and decides on the actual PDs according to its own semantics. For example, an `and` sub-constraint returns the union of the results of the child constraints, while an `or` sub-constraint returns the intersection of the results of the child constraints. Since new PDs are decided according to current ones, the process is iterative and stops when no more PDs are found. In our CSP example, we infer from the third constraint that the nine most-significant-bits of V1 are predetermined to be 0.

4.3 Dealiasing

The Dealiasing processing method finds and enforces *aliases*. An alias is a pair $(V, f(\mathcal{V}))$ of a variable and a function of other CSP variables, such that $V = f(\mathcal{V})$ in any solution to the CSP. Dealiasing limits the search space to assignments that satisfy the aliases.

V1,V2 [0,M];
constraints
1. (V1 = M) `or` (V1 = 0);
2. (V2 = M) `or` (V2 = 0);
3. (V1 = V2)

An alias can be inferred from a sub-constraint node of the form "V = EXP", where V is a variable and EXP an expression,[2] but only if the sub-constraint's path to the root node is composed only of `and` (or equivalent) operators. After collecting all the aliases that can be identified in the CSP, all the references to the aliased variables are replaced by references to the corresponding functions. Before describing the Dealiasing algorithm, let us illustrate the criticality of aliasing for SLS[3]. Consider the simple CSP in the above box, for some positive number M. A natural cost derived from the `and` operator at the root of the CSP, and `or` operators of constraints 1 and 2 is:

$$\text{Cost} = \min\left(||M - \text{V1}||, ||\text{V1} - 0||\right) + \min\left(||M - \text{V2}||, ||\text{V2} - 0||\right) + ||\text{V1} - \text{V2}||$$

where $||A - B||$ is the distance between A and B according to some defined metric. For any choice of a reasonable linear metric (e.g., absolute-value of difference, or Hamming distance) this cost induces huge plateaus in the search space. For example, all states for which V1 is closer to M than to 0, while V2 is closer to 0 are plateau states. This renders the problem, as formulated, hard for SLS.

After Dealiasing, the CSP contains two copies of the second constraint and the cost becomes $\text{Cost} = 2\min\left(||M - \text{V2}||, ||\text{V2} - 0||\right)$. This cost has two global minima, no local minima, and no plateaus.

The Dealiasing algorithm uses two sub-constraint node interfaces:

```
GetAliases()
ApplyAliases()
```

[2] An alias can also be inferred from a sub-constraint if it can be transformed to a functional form. For example V1 + V4 = 7 can be transformed to V1 = 7−V4.

[3] In contrast, Dealiasing hardly helps reach a solution in MAC-based algorithms because an aliased constraint of the form $V = f(\mathcal{V})$ will just propagate from $\mathcal{V}$ to V.

`GetAliases()` returns a list of all the aliases found in the sub-constraint: For example, an `and` sub-constraint returns the union of the lists returned by its children, while an `or` sub-constraint returns no aliases. `ApplyAliases()` replaces all occurrences of the aliased variable with a reference to the function to which the variable is aliased (possibly turning the CSP tree into a DAG).

The Dealiasing processing method is implemented by calling `GetAliases()` for the root node of the CSP tree to get a set of aliases A, finding a consistent subset of aliases $A1 \subseteq A$ (in order to avoid cyclic definitions between the aliases), and calling `ApplyAliases()` for the root node with $A1$[4]. In our CSP example we now infer that V2 is aliased to V3 from the second constraint. We replace all occurrences of V2 by V3 accordingly. The reformulated problem is now:

1. ((V3 $=$ V5+3) `and` (V5 $\leq$ 10)) or (V3 $\neq$ V3);
2. ((V4 $>$ 11) `and` (V3 $=$ V3));
3. ((V1 $<$ 5) or ((V1 $<$ 12) `and` (V1 $>$ V3$-$4)));

4.4 Pruning - Removing Tautologies and Contradictions

In the *prune* processing method, we recursively remove sub-constraints that are identified as tautological or contradictory. Tautological sub-constraints are ones that would be satisfied in any assignment consistent with known predetermined dimensions. Contradictory sub-constraints would be unsatisfied by any such assignment. We call both contradictory and tautological sub-constraints *redundant* sub-constraints. Removing redundant sub-constraints serves three purposes:

1. The cost function of the pruned CSP exhibits better FDC. For example, suppose that in a sub-constraint of the form "C1 `or` C2", C1 is contradictory. Then the natural cost function $\min(\text{Cost}(C1), \text{Cost}(C2))$ may exhibit a local minimum where Cost(C1) is minimized. Replacing "C1 or C2" by the equivalent "C2" immediately prevents this problem.
2. Creating more opportunities for inferences by other processing methods. In Section 4.5, we exemplify this effect on our CSP example.
3. Reducing the computational toll of calculating the cost function.

Though, in general, it has been shown that removing redundant constraints does not necessarily improve gradient solutions [18], in our experiments this has proved to be a vital step in complex expression-based problems. We attribute this to the combination of the three items listed above. These items may be less relevant to simple and well-structured CSPs. (Item 2 is only relevant to solvers applying our other processing methods.)

The following interface is implemented for any sub-constraint node type:

`Prune()`

[4] Finding a maximal set $A1$ is equivalent to the Directed Feedback Edge Set problem, which is NP-complete [17]. We therefore implement a heuristic algorithm that does not guarantee global maximality.

To run the pruning method, `Prune()` is called for the CSP tree root node. `Prune()` for an atomic sub-constraint may identify two kinds of redundant sub-constraints. First, it may identify patterns syntactically recognized as redundant, for instance "A > A", "A = A". (These patterns may exist in real-world CSPs that were generated automatically.) Second, it may identify constraints that are redundant due to constants and predetermined dimensions. The `Prune()` method for a compound sub-constraint may call `Prune()` for each of its child constraints and operate according to its own semantics. For example, the sub-constraint `and` would remove a tautological child node and would report itself as contradictory if one of its child nodes is contradictory. In our CSP example, the pruning method finds a contradiction and a tautology, resulting in:

```
1. (V3 = V5+3)  and  (V5 ≤ 10);
2. (V4 > 11);
3. (V1 < 5) or ((V1 < 12)  and  (V1 > V3−4)));
```

4.5 Combining Processing Methods

We apply an iterative algorithm to make full use of the interaction between the processing methods, stopping when no more changes occur.

```
Transform to NNF
repeat
  Find Predetermined Dimensions
  Apply Dealiasing
  Prune
until No changes have occurred in the last iteration
Find Unimportant Dimensions
```

This algorithm runs for two iterations on our CSP example. The first iteration results in the form given in Section 4.4. The second iteration then finds more predetermined dimensions in V5 and results in the alias (V3,V5+3). This allows another round of successful pruning. The final CSP is:

```
1. (V5 ≤ 10)
2. (V4 > 11);
3. (V1 < 5) or ((V1 < 12)  and  (V1 > (V5+3)−4)));
The 9 most-significant-bits of V1 and of V5 are predetermined to be 0.
```

This formulation is invariant to all processing methods. The unimportant dimensions are now inferred to be all bits of variables V2, V3 and V6, and the two unimportant bits of V4 found earlier.

5 Experimental Results

Experimental results were obtained using a tool called Stocs, which implements Simulated Variable Range Hopping (SVRH) [14]. Run-times reported include both the preprocessing and search phases, though the latter dominates in all cases not solved by preprocessing alone. The experiments were run on a single-core Intel (TM) 3GHz PC running Red-Hat Linux.

5.1 Artificial Example

Results for the CSP example are presented in the following table for several configurations of the processing methods. In the first column, results of using all processing methods are shown. In each of the other columns, one of the methods was disabled. Each scenario was run 10 times, with different starting states and random seeds. The timeout was 10 seconds.

	All Methods	No Predetermined Dimensions	No NNF transformation	No Dealiasing
Solved	**10**	**10**	0	9
Avg. Time (sec)	**0.10**	0.26	N/A	2.08
Min. Time (sec)	**0.08**	0.10	N/A	0.33
Max. Time (sec)	**0.12**	0.43	N/A	5.97

5.2 'Still Life' CSP

The table on the right lists results for the Still Life problem (prob0032 in CSPLib). The problem was modeled in a straightforward manner using the input language of Figure 1. To change it from an optimization problem to a CSP, a constraint requiring a minimum number of live cells was added. Each Life CSP was run 20 times with and without our processing methods, starting from different initial states and random seeds. Times are in seconds. The timeout was 500 seconds. The main processing method to affect the Still Life problem is the NNF transformation described in Section 4.1.

Board Size	Live Cells	Solved with Processing	Avg. Time	Solved w/o Processing	Avg. Time
6X6	14	100%	21	70%	188
6X6	15	95%	33	35%	245
6X6	16	100%	54	40%	241
6X6	17	100%	42	10%	294
6X6	18	100%	57	25%	290
7X7	24	95%	60	15%	355
7X7	25	75%	112	5%	356
7X7	26	70%	126	0	N/A
7X7	27	65%	139	0	N/A
7X7	28	65%	219	0	N/A
8X8	30	95%	78	0	N/A
8X8	32	75%	167	0	N/A
9X9	35	90%	152	0	N/A
9X9	37	70%	161	0	N/A
9X9	39	80%	138	0	N/A

5.3 Processor Verification

Most CSPLib problems are not natural candidates for testing our processing methods, as they have a simple recurring structure that can be easily modeled without requiring automatic processing to improve modeling. Industrial problems, on the other hand, can be very large and irregular, thus making it hard to manually model them in a way that would not hinder the results of an SLS solver. Moreover, the model is sometimes generated in a distributed fashion, making it even harder to take into account global considerations such as removing redundancies when modeling the CSP. Hardware verification [15] is one example of a domain where such problems are abundant. Our processing methods were applied to an industrial processor verification problem, generated automatically from user-interfaces for modeling the micro-architecture of the processor [19]. Typical problems consisted of 1,500 variables and 15,000 constraints. Details of the results of this work are beyond the scope of the paper, however, we can report that our processing methods reduced the number of variable-bits in the search space from about 100,000 to about 33,000, and enabled the SLS solver to reach significantly deeper local minima at a much shorter run-time.

5.4 Random Expression-Based CSPs

In order to test our algorithm in a less controlled environment, we generated a new type of random CSPs. Given a formal grammar of an input language such as the one shown in Figure 1, we consider ensembles of problems created by probabilistic formal grammar rules that generate a subset of the input language. A random expression-based problem is defined by $< N, D, M, G, \boldsymbol{p} >$, where N is the number of variables, D is the domains, M is the number of constraints, G is the formal grammar, and $\boldsymbol{p}$ is a probability-vector for the probabilistic rules. Like many real-world problems, random problems generated using this scheme exhibit little regularity despite the small number of parameters that control their creation, with constraints of varying tree-depth and complexity. [5]

Table 1. Probabilistic Grammar for Random CSP Generation

CONS $\longrightarrow$ (0.9) PC | (0.1) (not PC)
EXP $\longrightarrow$ (0.1) (EXP ARITH EXP) | (0.72) VAR | (0.08) NUM
PC $\longrightarrow$ (0.3) (CONS OP CONS) | (0.7) (EXP COMP EXP)
OP $\longrightarrow$ (0.25) and | (0.25) or | (0.25) implies | (0.25) iff
ARITH $\longrightarrow$ (0.33) + | (0.66) *
COMP $\longrightarrow$ (0.35) $>$ | (0.35) $<$ | (0.15) $=$ | (0.15) $\neq$
VAR $\longrightarrow$ Uniform probability over variables
NUM $\longrightarrow$ Uniform probability over domain range

We generated 300 random CSPs according to the probabilistic grammar of Table 1. Each CSP had 50 variables with domains $[0, 1000]$, and 20 constraints generated by the non-terminal CONS. Relatively large variable domains were chosen because such domains are characteristic of many industrial problems, such as verification [15] and workforce management [20]. Additionally, the problem is much harder to solve with larger domains. Making it harder by increasing the number of variables and constraints would make it cumbersome to analyze. The structure of the random expressions created by this particular grammar is reminiscent in many ways of the processor verification problems discussed above.

The solver was run 20 times on each CSP in 4 configurations. A total of 156 CSPs were solved at least once. For these, we compared solve-times between a

[5] A repository of CSP instances generated according to this scheme and used in this and the next sub-section can be found in `http://www.haifa.il.ibm.com/projects/verification/octopus/random`.

configuration that involves no processing methods, a configuration that includes all methods, and configurations that disable one method. We consider only statistically significant differences in solve-times[6]. Figure 4 shows the improvements in solve-times achieved for each CSP in the different configurations, compared to solving the CSP with no preprocessing. The following table summarizes the results, again comparing solve-times when using some preprocessing methods to using no preprocessing.

	Applying All Methods	No Predetermined Dimensions	No NNF transformation	No Dealiasing
Improvement	**79% (123)**	**69% (107)**	**39% (61)**	**40% (63)**
Deterioration	17% (26)	18% (28)	6% (9)	10% (16)
No Difference	4% (7)	13% (21)	55% (86)	50% (77)

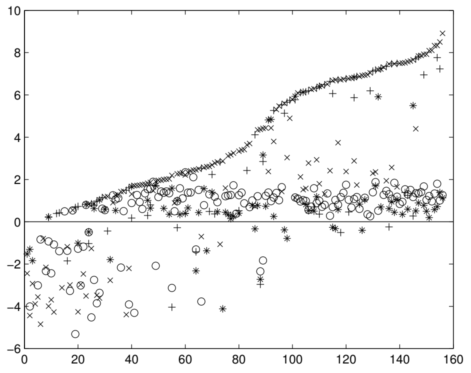


Fig. 4. Effect of the methods on random expression CSPs. The X axis corresponds to 156 CSPs, sorted by ascending ratio of improvement by our methods. The Y axis is log_2 of the ratio of time improvement by each configuration. Points above the zero line are ones in which improvement was achieved. Legend: circle = no predetermined dimensions, plus = no NNF-transformation, star = no Dealiasing, x = all methods applied

5.5 Phase Transition in Random Expression-Based Problems

It is well documented that random tabular CSP and random SAT problems exhibit satisfiability phase transitions [21, 22]. There exists a critical curve that describes the percentage of satisfiable instances of the random ensemble as a function of some parameter, e.g., the ratio between the number of constraints and the number of variables. Given a structure of the random problem, in the thermodynamic limit (i.e., with a large number of variables), the critical curve is universal — it does not depend on specifics such as the number of variables in the problem. Furthermore, instances with near-critical parameter values are found to be the hardest for systematic search.

We report a preliminary investigation of this phenomena on the random expression-based CSPs defined in the previous section. We used the grammar of Figure 1 with a probability-vector $\boldsymbol{p}$ to generate ensembles of 50 random problems for given numbers of variables N_V and constraints N_C. Figure 5 shows the percentage of problems solved by Stocs within a timeout of up to one minute.[7]

[6] A paired t-test with significance level of 0.05 was used.

[7] The timeout was set to a value much larger than the longest time for which a solution was ever found at a given combination of N_V and N_C.

A clear transition from solvable to unsolvable is observed as a universal function of N_C/N_V. We also find that the exact location of the transition depends on the probability-vector $\boldsymbol{p}$. As with systematic search, the hardest instances are around the critical area. Although the solver is not complete, it easily identifies unsatisfiable instances for CSPs far above the critical area during the application of its model-processing methods.

6 Summary

We presented a set of methods for processing a CSP model that is expressed using an expression language, in order to make this model more suitable for solving with an SLS solver. We used an architecture that allows definition of operators as part of an input expression language for SLS solvers. By implementing the abstract interfaces for a new operator, the CSP defined by those operators automatically benefits from the model-processing methods we introduced. This extends the clear separation between the model and the search algorithm to the pre-search phase.

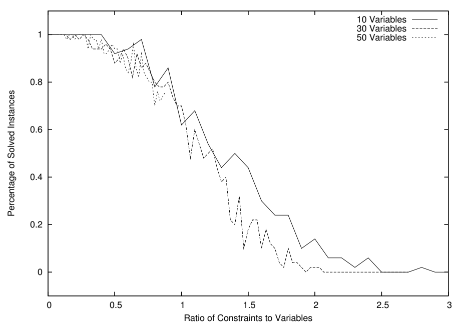


Fig. 5. Phase transition in expression based CSPs.

One part of the work that we only briefly investigated covers the random expressions introduced in sub-sections 5.4. We conjecture that this form of random CSPs resembles real world CSPs much better than random-table CSPs or random SAT instances. More specifically, by tuning the rules and parameters of the formal grammar, different ensembles may be generated, each possibly resembling a different application domain. Investigation of such ensembles may guide the design of algorithms and heuristics suitable for the particular domain.

7 Acknowledgments

We are grateful to Eyal Bin for presenting us with the processor verification problem and for defining much of the syntax of the expression language we used.

References

1. van Hentenryck, P.: The OPL optimization programming language. MIT Press, Cambridge, MA, USA (1999)
2. Hoos, H.H., Steutzle, T.: Stochastic Local Search: Foundations and Applications. Morgan Kaufmann, San Francisco (2004)

3. van Hentenryck, P., Michel, L.: Control abstractions for local search. In: CP 2003. (2003)
4. Nareyek, A.: Using global constraints for local search. In: Constraint Programming and Large Scale Discrete Optimization, DIMACS Vol. 57. (2001) 9–28
5. Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26. (1996)
6. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220** (1983) 671–680
7. Minton, S., Johnston, M., Phillips, A., Laird, P.: Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In: AAAI-90. (1990) 17–24
8. Glover, F., Laguna, M.: Tabu Search. Kluwer (1997)
9. Hansen, P., Mladenovic, N.: Introduction to variable neighbourhood search. In: Metaheuristics: Advances and Trends in Local Search Procedures for Optimization. (1999) 433–458
10. Bacchus, F.: Enhancing Davis Putnam with extended binary clause reasoning. In: AAAI-02. (2002)
11. Bacchus, F., Walsh, T.: Propagating logical combinations of constraints. In: IJCAI-05. (2005) 35–40
12. Boese, K.D., Kahng, A.B., Muddu, S.: A new adaptive multi-start technique for combinatorial global optimizations. Operations Res. Lett. **16**(3) (1994) 101–113
13. Bohlin, M.: Improving cost calculations for global constraints in local search. In: CP 2002. (2002) 772
14. Naveh, Y.: Stochastic solver for constraint satisfaction problems with learning of high-level characteristics of the problem topography. In: Local Search Techniques in Constraint Satisfaction (LSCS-04). (2004)
15. Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcus, E., Shurek, G.: Constraint-based random stimuli generation for hardware verification. AI Magazine (2007)
16. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. Lecture Notes in Computer Science **1520** (1998) 417
17. Garey, M., Johnson, D.: Computers and Intractability: a Guide to Theory of NP-completeness. W.H.Freeman (1979)
18. Hentenryck, P.V., Michel, L.: Constraint-Based Local Search. The MIT Press (2005)
19. Adir, A., Bin, E., Peled, O., Ziv, A.: Piparazzi: A test program generator for micro-architecture flow verification. In: Eighth IEEE International High-Level Design Validation and Test Workshop, HLDVT-03. (2003) 23–28
20. Naveh, Y., Richter, Y., Altshuler, Y., Gresh, D.L., Connors, D.P.: Workforce optimization: Identification and assignment of professional workers using constraint programming. IBM Journal or Research and Development (2007)
21. Prosser, P.: An empirical study of phase transition in binary constraint satisfaction problems. Artificial Intelligence **81** (1996) 81–109
22. Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., Ttroyansky, L.: Determining computational complexity from characteristic 'phase transition'. Nature **400** (1999) 133–137