A Meta-Heuristic for Subset Decision Problems

Brahim Hnich, Zeynep Kızıltan, and Pierre Flener
Computer Science Division, Department of Information Science
Uppsala University, Box 513, S–751 20 Uppsala, Sweden
{Brahim.Hnich, Zeynep.Kızıltan, Pierre.Flener}@dis.uu.se

1 Introduction

Constraint Satisfaction Problems (CSPs) — where appropriate values for the variables of the problem have to be found, subject to some constraints — represent many real life problems. Examples are production planning subject to demand and resource availability, air traffic control subject to safety protocols, transportation scheduling subject to initial and final location of the goods and the transportation resources, etc. Many of these problems can be expressed as constraint programs and then be solved using constraint solvers.

Most of the available constraint solvers (clp(FD) [1], OPL [13], etc) are equipped with constraint propagation algorithms based on consistency techniques such as node and arc consistency, plus a search algorithm such as forward-checking, and a labeling heuristics, one of which is the default. To enhance the performance of a constraint program, a lot of research has been made in recent years to develop new heuristics concerning the choice of the next variable to branch on during the search and the choice of the value to be assigned to that variable, giving rise to variable and value ordering (VVO) heuristics. These heuristics significantly reduce the search space [9]. However, little is said about the application domain of these heuristics, so programmers find it difficult to decide when to apply a particular heuristic and when not.

The difficulty of mapping the right heuristic to a given problem is mainly due to two reasons. First, as mentioned by Tsang *et al.* [10], there is no universally best heuristic for all problems. Thus, we are only able to learn that a particular heuristic is best for the particular benchmarks used by researchers to carry out their experiments. Second, as noticed by Minton [8], the performance of heuristics is instance-dependent, i.e., for a given problem a heuristic can perform well for some distributions on the instances, but very poorly on other distributions.

To understand our terminology, note that the phrase *problem class* here refers to a whole set of related problems, while the term *problem* designates a particular problem (within a class), and the word *instance* is about a particular occurrence of a problem. For example, planning is a problem class, traveling salesperson is a problem within that class, and visiting all nodes of the ERCIM Working Group on Constraints is an instance of that problem. Much of (constraint) programming research is about pushing results from the instance level to the problem level if not to the problem-class level, so as to get reusable generic approaches.

We here use constraint solvers as blackboxes, thus fixing the propagation and search algorithms, while trying to find an appropriate VVO (meta-)heuristic that performs at least better than the default one. To illustrate our approach, we focus on a particular problem class, namely subset decision problems. Assuming that we have an initial set $\mathcal{H}$ of VVO heuristics (including the default one), we take an empirical approach to find a meta-heuristic that can decide which heuristic in $\mathcal{H}$ best suits the instance to be solved. Such a meta-heuristic can then be integrated within the constraint solver.

This paper is organised as follows. In Section 2, we discuss the class of subset decision problems and show the generic clp(FD) constraint store that results from such problems. Then, in Section 3, we present our empirical approach, show our results, and explain the usage of our meta-heuristic for subset decision problems. Finally, in Section 4, we conclude, compare with related work, and discuss our directions for future research.

2 Subset Decision Problems

We assume that CSP models are initially written in a very expressive, purely declarative, typed, first-order set constraint logic programming language, such as our proposal in [3], here called ESRA, which is being designed to be higher-level than even OPL [13]. Using program synthesis techniques such as those in [11, 8, 2], we can automatically compile ESRA programs into lower-level languages such as clp(FD) or OPL. The purpose of this paper is not to discuss how this can be done, nor the syntax and semantics of ESRA.

In the class of *subset decision problems*, a subset S of a given finite set T has to be found, such that S satisfies an (open) condition g, and an arbitrary two different elements of S satisfy an (open) condition p. In ESRA, we model this as the following (open) program:

$$\begin{gathered}\forall T, S : set(int)\,.\\ subset(T,S) \leftrightarrow S \subseteq T \wedge g(S) \wedge\\ \forall I, J : int\,.\, I \in S \wedge J \in S \wedge I \neq J \rightarrow p(I,J)\end{gathered} \tag{subset}$$

The only open symbols are relations g and p (assuming that $\subseteq$, $\in$, and $\neq$ are primitives of ESRA, with the usual meanings). This program has as refinements programs for many problems, such as finding a clique of a graph (see below), set covering, knapsack, etc. For example, the (closed) program:

$$\begin{gathered}\forall V, C : set(int)\,.\,\forall E : set(int \times int)\,.\\ clique20(\langle V, E\rangle, C) \leftrightarrow C \subseteq V \wedge size(C, 20) \wedge\\ \forall I, J : int\,.\, I \in C \wedge J \in C \wedge I \neq J \rightarrow \langle I, J\rangle \in E\end{gathered} \tag{clique20}$$

is a refinement of *subset*, under the substitution:

$$\begin{gathered}\forall C : set(int)\,.\, g(C) \leftrightarrow size(C, 20)\\ \\ \forall E : set(int \times int)\,.\\ \forall I, J : int\,.\, p(I,J) \leftrightarrow \langle I, J\rangle \in E\end{gathered} \tag{σ}$$

assuming that *size* is another primitive of ESRA, with the obvious meaning. It is a program for a particular case of the *clique problem*, namely finding a clique (or: a maximally connected component) of an undirected graph (which is given through its vertex set V and its edge set E), such that the size of the clique is 20.

At a lower level of expressiveness, subset decision problems can be compiled into clp(FD) constraint programs, say. The chosen representation of a subset S of a given *finite* set T (of n elements) is a mapping from T into Boolean values (domain variables in $\{0,1\}$), that is we conceptually maintain n couples $\langle I, B_I\rangle$ where the (initially non-ground) Boolean B_I expresses whether the (initially ground) element I of T is a member of S or not:[1]

$$\forall I : int\,.\, I \in T \rightarrow (B_I \leftrightarrow I \in S) \tag{1}$$

This Boolean representation of sets consumes more memory than the set interval representation of CONJUNTO [6] and OZ, but both have been shown to create the same search space [6]; moreover, the set interval representation does not allow the definition of some (to us) desirable high-level primitives, such as universal quantification over elements of non-ground sets. (Another alternative representation of the subset S, namely as a sequence of k ($\leq n$) variables constrained to be different elements of T, has two disadvantages compared to ours: first, the search space for S then is much worse, namely $O(n!)$, and second, an explicit loop for k ranging from 0 to n has to be wrapped around the code.)

Given this Boolean representation choice for sets, the formula for the open relation g of *subset* can easily be re-stated in terms of constraints on Boolean variables. As shown in [3], it is indeed easy to write constraint-*posting* clp(FD) programs for $\in$, $\subseteq$, *size*, and all other classical set operations. We here pay special attention to the case where g (also) constrains the size of the subset to be a constant, say k. This can be written as the following constraint:

$$\sum_{i=1}^{n} B_i = k \tag{2}$$

Let us now look at the remaining part of *subset*, which expresses that any two different elements of the subset S of T must satisfy a condition p:

$$S \subseteq T \wedge \forall I, J : int\,.\, I \in S \wedge J \in S \wedge I \neq J \rightarrow p(I,J)$$

This statement can be refined as follows:

$$\begin{gathered}\forall I, J : int\,.\, I \in T \wedge J \in T \wedge\\ I \in S \wedge J \in S \wedge I \neq J \rightarrow p(I,J)\end{gathered}$$

which is equivalent to:

$$\begin{gathered}\forall I, J : int\,.\, I \in T \wedge J \in T \wedge I \neq J \wedge \neg p(I,J)\\ \rightarrow\ \neg(I \in S \wedge J \in S)\end{gathered}$$

By (1), this can be rewritten as:

$$\begin{gathered}\forall I, J : int\,.\, I \in T \wedge J \in T \wedge I \neq J \wedge \neg p(I,J)\\ \rightarrow\ \neg(B_I \wedge B_J)\end{gathered}$$

Thus, for every two distinct elements I and J of T, with corresponding Boolean variables B_I and B_J, if $p(I,J)$ does not hold, we just need to post the constraint $\neg(B_I \wedge B_J)$.

Note that the posted clp(FD) constraints are thus *not* in terms of p, hence p can be *any* ESRA formula and our approach works for the *whole* class of subset decision problems. Indeed, the reasoning above was made for the (open) *subset* program rather than for a particular (closed) refinement such as *clique*20.

Therefore, the clp(FD) constraint store for *any* subset decision problem is over a set of Boolean variables and contains an instance-dependent number of binary constraints of the form $\neg(B_I \wedge B_J)$ (if p is not *true*) as well as an optional summation constraint (2) (if g also uses *size*). All other constraints in g are (currently) ignored in our quest for a meta-heuristic.

3 A Meta-Heuristic for Subset Decision Problems

We now present our approach for devising a meta-heuristic for the entire class of subset decision problems. On the one hand, as shown in the previous section, we are able to map all subset decision problems into a generic clp(FD) constraint store, depending on the number n of Boolean variables involved (i.e., the size of the given set), the optional subset size k, and the number of binary constraints b. On the other hand, an ever increasing set $\mathcal{H}$ of VVO heuristics for CSPs is being proposed. Our approach now is

[1] In formulas, we use atom B_I as an abbreviation for $B_I = 1$.

to first measure the run-time of each heuristic, for a fixed clp(FD) solver, on a large number of instances with different values for n, k, and b. Then we try and determine the range (in terms of n, k, and b) for every heuristic in which it performs best, so as to implement a meta-heuristic that always picks the best heuristic in $\mathcal{H}$ for any instance.

To illustrate the idea, let us assume that we have two heuristics, H_1 and H_2 say. If we keep n and b constant, we can measure the run-times of both heuristics for all values of k. The plot in Figure 1 suggests the following meta-heuristic:

$$\begin{array}{l} \text{if } k \in 1..3 \text{ then choose } H_1 \\ \text{if } k \in 3..5 \text{ then choose } H_2 \\ \text{if } k \in 5..n \text{ then choose } H_1 \end{array}$$

However, in our case, the problem is more difficult because we have 3 varying dimensions rather than just 1, namely n, k, and b.

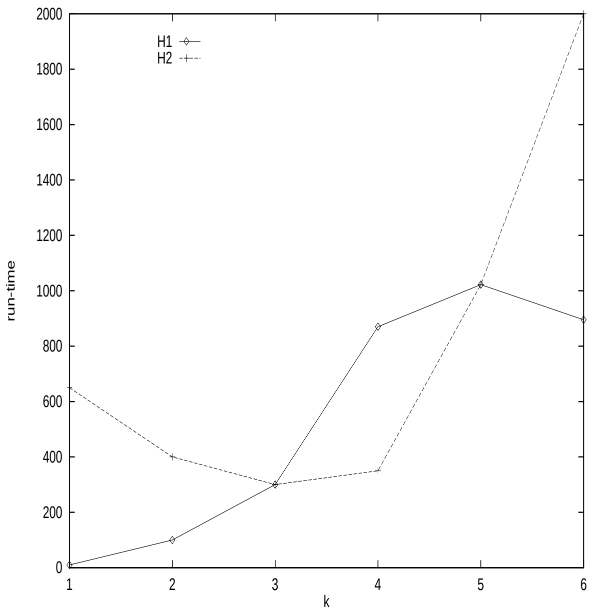

Figure 1: Run-time in terms of k for 2 heuristics.

We now introduce our experimental setting and results, and then show how to use those results to devise a meta-heuristic for subset decision problems.

3.1 Experimental Setting and Results

For the purposes of this paper, we focused on 3 VVO heuristics only, as we would first like to show that the principle works. More VVO heuristics can easily be added to the experiments, if given more time. We also generated random instances in a coarse way (by not considering all possible combinations of n, k, and b); again, given more time, instances generated in a more fine-grained way could be used instead and help make our results more precise.

We used the following 3 VVO heuristics:

- The *default VVO heuristic* labels the leftmost variable in the sequence of variables provided, and the domain of the chosen variable is explored in ascending order.

- The *static VVO heuristic* pre-orders the variables in ascending order, according to the number of constraints in which a variable is involved, and then labels the variables according to that order by assigning the value 1 first (recall that we need only consider the Boolean domain $\{0, 1\}$).

- The *dynamic VVO heuristic* is based on the one described by Geelen [4]; however, the variable is here chosen in a way that maximises the sum of the promises of its values, and it is labeled with the least promising value. Although this max-min heuristic is the opposite of the min-max heuristic advocated by Geelen [4], our experiments showed that it performed best for subset decision problems, among all the 4 alternatives for variable and value ordering.

The default VVO heuristic does not introduce any extra overhead. The static one has a pre-processing overhead, while the dynamic one is the most costly one, as it incorporates calculations at each labeling step. We tested the effect of these heuristics by using the same propagation and search algorithms, namely the ones of SICSTUS CLP(FD).

Instance Generation

As described in Section 2, the clp(FD) constraint store for *any* subset decision problem is over a set of Boolean variables and contains an instance-dependent number of binary constraints as well as an optional summation constraint. For binary CSPs, instances are characterised by a tuple $\langle n, m, p_1, p_2 \rangle$ [10], where n is the number of variables, m is the (constant) domain size for all variables, p_1 is the constraint density,[2] and p_2 is the tightness of the individual constraints.

In our experiments, the domain size m is fixed to 2 as we need only consider the Boolean domain $\{0, 1\}$ in subset decision problems. The number n of variables ranged over the interval 10..200, by increments of 10. We varied the values of p_1 over the interval 0.1..1, by increments of 0.1. Since the considered binary constraints are of the form $\neg(B_I \wedge B_J)$, their tightness is always equal to 3/4 and they can thus be ignored in the computation of p_2. Therefore, only the summation constraint determines p_2; its tightness, and therefore the tightness of all the considered constraints, is:

$$p_2 = \frac{\binom{n}{k}}{2^n}$$

Instead of varying the values of p_2, we varied the values of k, over the interval $1..\lfloor n/2 \rfloor$, by increments

[2] Note that $p_1 = \frac{b}{n(n-1)/2}$.

of 1, as this also leads to an interval of p_2 values, since n ranges over an interval. (In any case, varying p_2 by a constant increment over the interval 0..1 would have missed out on a lot of values for k. Indeed, when k ranges over the integer interval above, the corresponding values of p_2 do not exhibit a constant increment within 0..1.) The chosen upper bound of the interval for k is sufficiently big because of the symmetric nature of combinations.

Experiments and Results

Having thus chosen the intervals (and increments) for the parameters describing the characteristics of instances of subset decision problems, we randomly generated many different instances and then used the 3 chosen heuristics in order to solve them. Note that not every instance has a solution. Also, some of the instances were obviously too difficult to solve within a reasonable amount of time. Consequently, to save time in our experiments, we used a time-out on the CPU time; hence, our meta-heuristic can currently not select the best heuristic for a given instance characterisation when all 3 heuristics were timed out on it. The obtained results are tabulated as $\langle n, p_1, k, t_1, t_2, t_3 \rangle$ tuples, where t_i is the CPU time for heuristic i:

n	p_1	k	t_1	t_2	t_3
⋮	⋮	⋮	⋮	⋮	⋮
100	0.2	6	40	970	2030
⋮	⋮	⋮	⋮	⋮	⋮
110	0.2	22	time out	20	1880
⋮	⋮	⋮	⋮	⋮	⋮
130	0.3	18	time out	10250	5150
⋮	⋮	⋮	⋮	⋮	⋮

We can see that indeed no heuristic outperforms all other heuristics, or is outperformed by all other heuristics. Moreover, the collected run-times look very unpredictable and have many outlyers. This confirms Minton's and Tsang *et al.*'s results and also shows that human intuition breaks down here (especially when dealing with blackbox solvers).

In order to analyse the effects of each heuristic on different instances, we drew various charts, for example by keeping n and p_1 constant and plotting the run-times for each k. Figure 2 shows an example of the behaviours of the 3 heuristics on the instances where $n = 110$ and $p_1 = 0.4$.

3.2 Usage of the Results

Using the obtained table as a lookup table, it is straightforward to devise a (static) meta-heuristic that first measures the parameters $\langle n, p_1, k \rangle$ of the given instance, and then uses the (nearest) corresponding entry in the table to determine which

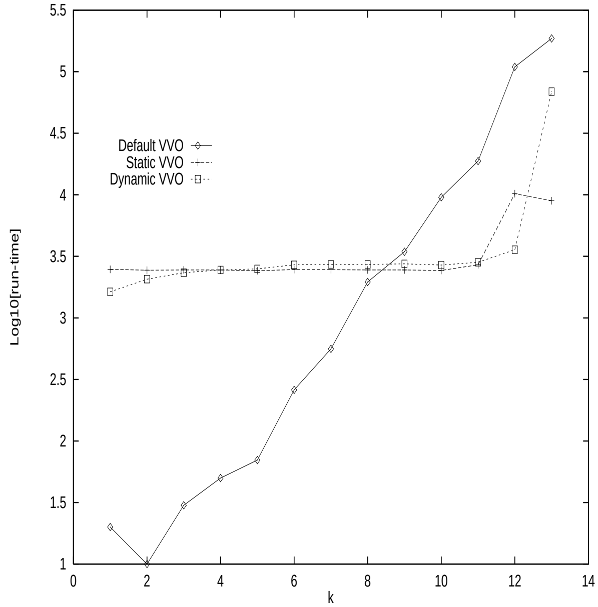

Figure 2: Run-time in terms of k for the 3 heuristics on $n = 110$ and $p_1 = 0.4$.

heuristic to actually run on this instance. Considering the simplicity of these measures, the (constant) run-time overhead is negligible, especially that it nearly always pays off anyway. The meta-heuristic (including the table) and the code of all involved heuristics thus become part of the generated instance-independent program, but it is guaranteed to make the program run, for *any* instance, (almost exactly) as fast as the fastest heuristic for that instance.

From the results of the empirical study, we can also conclude the following, regarding subset decision problems:

- As instances get less constrained [5], the default VVO heuristic almost always performs best.
- As instances get more constrained, the performance of the default VVO heuristic degenerates (see Figure 3).
- As instances get more constrained, the static and dynamic VVO heuristics behave much more gracefully, rather than seeing their run-times degenerate (see Figure 3).
- Even though it is very costly to calculate the dynamic VVO heuristic, it sometimes outperforms the other two heuristics.
- For some of the instances, all the heuristics failed to find a solution within a reasonable amount of time.

4 Conclusion

We have shown how to map an entire class of CSPs, namely subset decision problems, to a generic

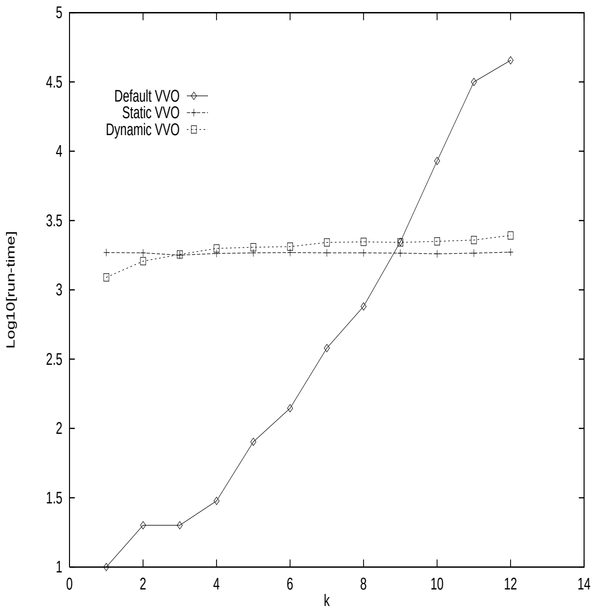

Figure 3: Run-time in terms of k for the 3 heuristics for $n = 100$ and $p_1 = 0.4$.

clp(FD) constraint store, and we have devised a class-specific but problem-independent meta-heuristic that chooses an instance-specific heuristic that is guaranteed to perform as well as the best considered heuristic, for any instance. This work is thus a continuation of Tsang *et al.*'s research [10] on mapping heuristics to application domains, and an incorporation of Minton's and Tsang *et al.*'s findings about the sensitivity of heuristics to instance distributions. The key insight is to analyse and exploit the form (and number) of the actually posted constraints for a problem class, rather than considering the constraint store a black box and looking for optimisation opportunities elsewhere.

The importance and contribution of this work is to have shown that some form of heuristic, even if "only" a meta-heuristic, and a brute-force one at that, *can* be devised for an entire problem class, without regard to its problems or their instances. Considering the availability and automatic selection by a solver of such a (meta-)heuristic, programmers can be encouraged to model CSPs as subset problems rather than in a different way (if the possibility arises at all). Indeed, they then do not have to worry about which heuristic to choose, nor do they have to implement it, nor do they have to document the resulting program with a disclaimer stating for which distribution of instances it will run best. All these non-declarative decisions can thus be taken care of by the solver, leaving only the declarative issue of modeling the CSP to the programmers, thus extending the range and size of CSPs that they can handle properly. Further advances along these lines will bring us another step closer to the holy grail of programming (for CSPs).

4.1 Related Work

This work follows the call of Tsang *et al.* for mapping combinations of algorithms and heuristics to application domains [10]. However, we here focused on just one application domain (or: class of problems), as well as on just the effect of VVO heuristics while keeping the algorithm constant.

Also closely related to our work is Minton's MULTI-TAC system [8], which automatically synthesises an instance-distribution-specific program (i.e., algorithm and heuristic) for solving a CSP, given a high-level description thereof and a set of training instances (or an instance generator). His motivation also was that heuristics depend on the distribution of instances. However, we differ from his approach in various ways:

- While the performance of MULTI-TAC's synthesised programs is highly dependent on the distribution of the given training instances, we advocate the off-line brute-force approach of generating all possible distributions for given problem classes and analysing them towards the identification of suitable meta-heuristics.
- While MULTI-TAC uses a synthesis-time brute-force approach to generate candidate problem-and-instance-distribution-specific heuristics, we only choose our heuristics from already published ones.
- While it is the responsibility of a MULTI-TAC user to also provide training instances (or an instance generator plus the desired distribution parameters) in order to synthesise an instance-distribution-specific program, our meta-heuristic can be pre-computed once and for all, in a problem-independent way for an entire class of problems, and the user thus need not provide more than a high-level problem description.

Finally, the work of Smith *et al.* on the KIDS program synthesiser and its successors [11, 12] has some influence on ours. Their semi-automatic systems excel at generating (sometimes novel) programs for CSPs, though without any explicit recourse to constraint programming technology. Indeed, they synthesise *ad hoc* code given a high-level description of a CSP and a formal domain theory. By replacing their target language with clp(FD), we have been able to considerably reduce the need for their (computer-assisted) optimisation of the thus synthesised programs [2].

4.2 Future Work

Our plans for future work include investigating the possibility of devising a *dynamic* meta-heuristic that chooses a (possibly different) heuristic after each labeling iteration, based on the current sub-problem, rather than sticking to the same initially chosen static heuristic all the way. The hope is that the performance would increase even more, but this intuitively looks unlikely, as many heuristics look deeply ahead and thus only pick up speed after some slow first iterations, so that it would be counter-productive to then switch to another heuristic that starts all over. However, we have some ideas how to go at this.

We will furthermore try to derive an evaluation function (by regression analysis) instead of using the full look-up table. This would not speed up the resulting programs, but their size would shrink dramatically, as the look-up table would not have to be trailed around.

Of course, we should also produce instances in a more fine-grained way (over *all* $\langle n, p_1, k \rangle$ triples until some n) and involve more known heuristics, so as to further improve our meta-heuristic. This is just a matter of having the (CPU) time to do so.

The here studied class of subset decision problems can be generalised into the class of k-subset decision problems (where k subsets of a given set have to be found, subject to some constraints) [7]. Another extension is the coverage of (k-)subset *optimisation* problems. We expect to address these issues.

Finally, we are planning to investigate other classes of problems, namely *assignment problems* (where a mapping between two given sets has to be found, subject to some constraints) [2], *permutation problems* (where a sequence representing a permutation of a given set has to be found, subject to some constraints) [2], and *sequencing problems* (where sequences of (given or bounded) size over the elements of a given set have to be found, subject to some constraints).

References

[1] Ph. Codognet and D. Diaz. Compiling constraints in clp(FD). *J. of Logic Programming* 27(3):185–226, 1996.

[2] P. Flener, H. Zidoum, and B. Hnich. Schema-guided synthesis of constraint logic programs. In *Proc. of ASE'98*, pp. 168–176. IEEE Computer Society Press, 1998.

[3] P. Flener, B. Hnich, and Z. Kızıltan. Towards schema-guided compilation of set constraint programs. In B. Jayaraman and G. Rossi (eds), *Proc. of DPS'99*, pp. 59–66. Tech. Rep. 200, Math. Dept., Univ. of Parma, Italy, 1999.

[4] P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proc. of ECAI'92*, pp. 31–35. 1992.

[5] I.P. Gent, E. MacIntyre, P. Prosser, B.M. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proc. of CP'96*, pp. 179–193. The MIT Press, 1996.

[6] C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints* 1(3):191–244, 1997.

[7] B. Hnich and Z. Kızıltan. Generating programs for k-subsets problems. In P. Alexander (ed), *Proc. of the ASE'99 Doctoral Symposium*. 1999.

[8] S. Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints* 1(1–2):7–43, 1996.

[9] E.P.K. Tsang. *Foundation of Constraint Satisfaction*. Academic Press, 1993.

[10] E.P.K. Tsang, J.E. Borret, and A.C.M. Kwan. An attempt to map the performance of a range of algorithm and heuristic combinations. In *Proc. of AISB'95*, pp. 203–216. IOS Press, 1995.

[11] D.R. Smith. KIDS: A semi-automatic program development system. *IEEE Trans. on Software Engineering* 16(9):1024–1043, 1990.

[12] D.R. Smith. Toward a classification approach to design. *Proc. of AMAST'96*, pp. 62–84. LNCS 1101. Springer-Verlag, 1996.

[13] P. Van Hentenryck. *The* OPL *Optimization Programming Language*. The MIT Press, 1999.