A Meta-Heuristic for Subset Problems

Pierre Flener, Brahim Hnich, and Zeynep Kızıltan

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

Abstract. In constraint solvers, variable and value ordering heuristics are used to finetune the performance of the underlying search and propagation algorithms. However, few guidelines have been proposed for when to choose what heuristic among the wealth of existing ones. Empirical studies have established that this would be very hard, as none of these heuristics outperforms all the other ones on all instances of all problems (for an otherwise fixed solver). The best heuristic varies not only between problems, but even between different instances of the same problem. Taking heed of the popular dictum "If you can't beat them, join them!" we devise a practical meta-heuristic that automatically chooses, at run-time, the "best" available heuristic for the instance at hand. It is applicable to an entire class of NP-complete subset problems.

1 Introduction

If you can't beat them, join them!
— Anonymous

Constraint Satisfaction Problems (CSPs) — where appropriate values for the problem variables have to be found within their domains, 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 vehicles, etc. Many of these problems can be expressed as constraint programs and then be solved using constraint solvers.

Constraint solvers (such as SICSTUS CLP(FD) [2] and OPL [18]) are equipped with constraint propagation algorithms based on consistency techniques such as bounds consistency, plus a search algorithm such as forward-checking, and labelling 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 [16]. 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.

In order 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. (We here identify problems with their chosen models.) For example, planning is a problem class, travelling salesperson is a problem within that class, and visiting all capital cities of Europe 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 generic results.

The difficulty of mapping the right heuristic to a given problem is mainly due to the following. As shown by Tsang *et al.* [17], there is no universally best solver for all instances of all problems. Thus, we are only told that a particular solver is "best" for the particular instances used by researchers to carry out their experiments. Therefore, as also noticed by Minton [14], the performance of solvers is instance-dependent, i.e., for a given problem a solver can perform well for some (distributions on the) instances, but very poorly on others.

In such a case, conventional wisdom suggests joining the competitors, although we propose a novel way of interpreting this popular dictum: rather than joining efforts *with* the competitors (by teaming up with some of them), we advocate joining the efforts *of* the competitors, thus "acquiring" some of them and being at the helm! But, how can this be done here, as a solver cannot know in what situation it is? The answer is to do the investigation at the level of problem classes, and to enrich the solver accordingly.

Assuming that we have a set $\mathcal{H}$ of VVO heuristics (including the default one), we take an empirical approach to completely pre-determine a meta-heuristic that can decide which available heuristic in $\mathcal{H}$ "best" suits the instance to be solved, and this for *any* instance of *any* problem of the considered class. (We here use constraint solvers as blackboxes, thus fixing the propagation and search algorithms.) Such a meta-heuristic can then be added to the constraint solver. We here illustrate our approach with an NP-complete class of subset problems.

This paper[1] is organised as follows. In Section 2, we discuss a class of subset problems and show the generic finite domain constraint store that results from such problems. Then, in Section 3, we present our empirical approach and devise our meta-heuristic for subset 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, set-oriented constraint programming language, such as our ESRA [6], which is designed to be higher-level than even OPL [18]. We can automatically compile ESRA programs into lower-level finite-domain constraint 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.

[1] This paper is an extension of the unrefereed [11].

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) constraint g, and an arbitrary two distinct elements of S satisfy an (open) constraint p. In ESRA, we model this as (a sugared version of) the following (open) program:

$$\begin{array}{c} \forall T, S : set(\alpha)\,. \\ subset(T,S) \leftrightarrow S \subseteq T \land g(S) \land \\ \forall i,j : \alpha\,.\, i \in S \land j \in S \land i \neq j \rightarrow p(i,j) \end{array} \qquad (subset)$$

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

$$\begin{array}{c} \forall V, C : set(\alpha)\,.\, \forall E : set(\alpha \times \alpha)\,. \\ clique_5(\langle V,E\rangle, C) \leftrightarrow C \subseteq V \land size(C,5) \land \\ \forall i,j : \alpha\,.\, i \in C \land j \in C \land i \neq j \rightarrow \langle i,j\rangle \in E \end{array} \qquad (clique_5)$$

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

$$\begin{array}{c} \forall C : set(\alpha)\,.\, g(C) \leftrightarrow size(C,5) \\ \forall E : set(\alpha \times \alpha)\,.\, \forall i,j : \alpha\,.\, p(i,j) \leftrightarrow \langle i,j\rangle \in E \end{array} \qquad (\sigma)$$

where *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 edge set E), such that the size of the clique is 5.

At a lower level of expressiveness, ESRA subset problems can be compiled into finite-domain constraint programs. The chosen representation of a subset S of a given *finite* set T (of n elements) is a mapping from T into Boolean variables (in $\{0,1\}$), that is we conceptually maintain n couples $\langle T_i, B_i\rangle$ where the (initially non-ground) Boolean B_i expresses whether the (initially ground) element T_i of T is a member of S or not: [2]

$$\forall T_i : \alpha\,.\, T_i \in T \rightarrow (B_i \leftrightarrow T_i \in S) \qquad (1)$$

This Boolean representation of sets is different from the set interval representation of CONJUNTO [8] and OZ [15], but both have been shown to create the same $O(2^n)$ search space [8]. 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.

Given this Boolean representation choice for sets, the open constraints g and p of *subset* can easily be re-stated in terms of finite-domain constraints on Boolean variables. As shown in [5], it is indeed easy to write constraint-posting programs for $\in$, $\subseteq$, *size*, and all other classical set operations.

We here pay special attention to the NP-complete class where g only constrains the size of the subset to be a certain constant, and where p is not *true*.[3]

[2] In formulas, atom B_i is an abbreviation for atom $B_i = 1$.

[3] When g is *true*, subset problems can be reduced, in our representation, to 2-SAT (satisfiability of sets of 2-literal clauses), which is in P.

Having a good (meta-)heuristic for one NP-complete problem Q is of great practical interest, as some other NP problems can be reduced, in polynomial time and without loss of properties, to Q, so that the (meta-)heuristic is also usefully applicable to them.

Restricting the size of the subset to be a given constant, say k, can be written as the following n-ary 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 distinct elements of the subset S of T must satisfy a constraint p:

$$S \subseteq T \wedge \forall T_i, T_j : \alpha \,.\, T_i \in S \wedge T_j \in S \wedge T_i \neq T_j \rightarrow p(T_i, T_j)$$

This implies:

$$\forall T_i, T_j : \alpha \,.\, T_i \in T \wedge T_j \in T \wedge T_i \in S \wedge T_j \in S \wedge T_i \neq T_j \rightarrow p(T_i, T_j)$$

which is equivalent to:

$$\forall T_i, T_j : \alpha \,.\, T_i \in T \wedge T_j \in T \wedge T_i \neq T_j \wedge \neg p(T_i, T_j) \rightarrow \neg(T_i \in S \wedge T_j \in S)$$

By (1), this can be rewritten as:

$$\forall T_i, T_j : \alpha \,.\, T_i \in T \wedge T_j \in T \wedge T_i \neq T_j \wedge \neg p(T_i, T_j) \rightarrow \neg(B_i \wedge B_j)$$

Thus, for every two distinct elements T_i and T_j of T, with corresponding Boolean variables B_i and B_j, if $p(T_i, T_j)$ does not hold, we just need to post the following binary constraint:

$$\neg(B_i \wedge B_j) \tag{3}$$

It is crucial to note that the actually posted finite-domain constraints are thus *not* in terms of p, hence p can be *any* ESRA formula and our approach works for our *whole* class of subset problems! Indeed, the reasoning above was made for the (open) *subset* program rather than for a particular (closed) refinement such as $clique_5$.

Therefore, the finite-domain constraint store for *any* subset problem of the considered class is over a set of (only) Boolean variables and contains an instance-dependent number of binary constraints of the form (3) (if p is not *true*) as well as a summation constraint of the form (2).[4] Our results are thus “best” applied only in the NP class where p is not *true* and g is only a size constraint. Extending our results to other contents of g is only a matter of time, but the considered class is NP-complete and thereby our results are already significant.

[4] Having obtained what is largely a binary CSP (over Boolean variables) is a mere coincidence, and irrelevant to our approach.

3 A Meta-Heuristic for Subset Decision Problems

We now present our approach for devising a meta-heuristic for the entire presented class of subset problems, describe the used experimental setting, and show how to use the obtained results to devise a meta-heuristic for that class.

3.1 Approach Taken

On the one hand, we are able to map all subset problems of the considered class into a generic finite-domain constraint store, parameterised by the number n of Boolean variables involved (i.e., the size of the given set), the subset size k, and the number b of binary constraints of the form (3).[5] On the other hand, an ever increasing set $\mathcal{H}$ of VVO heuristics for CSPs is being proposed.

Our approach is to first measure the median cost (in CPU time and in number of backtracks) of each heuristic, for a fixed finite-domain solver, on a large number of instances with different values for $\langle n, k, b\rangle$. Then we try and determine the range of instances (in terms of $\langle n, k, b\rangle$) for every heuristic in which it performs "best," so as to devise a meta-heuristic that always picks the "best" heuristic in $\mathcal{H}$ for any instance. Note that these measures, and hence the meta-heuristic, are thus *entirely* made off-line and *once-and-for-all*, for *all* instances of *all* problems of the *whole* (and large) subset problem class. Compared to a hardwired choice of a heuristic, the run-time overhead of the meta-heuristic for a particular instance will *only* consist of counting the number b of actually posted binary constraints of the form (3) and then looking up which heuristic to use. For all but the most trivial instances, this overhead is negligible, because the calculations are easy.

Our approach rests on the assumption that all instances of the same $\langle n, k, b\rangle$ family[6] will benefit from the heuristic chosen for the instance that had the median cost. More analytical and empirical work is of course needed to better understand and model the variance in behaviour inside a family, and to understand whether $\langle n, k, b\rangle$ is an effective characterisation of subset problem instances.

To illustrate the idea, let us assume that we have just two heuristics, H_1 and H_2 say. If we keep n and b constant, we can measure the costs of both heuristics for all values of k. The illustrative plot (from made-up data) 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 three varying dimensions rather than just one, namely n, k, and b.

[5] It is the instance data that determine, after posting all constraints, the value of b.

[6] A family (of instances) is not to be confused with a class (of problems).

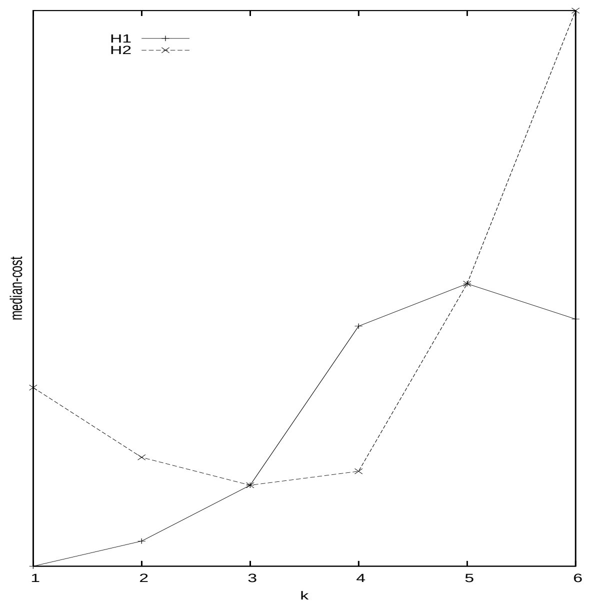

Fig. 1. Typical median-cost curve in terms of k for two heuristics

3.2 Experimental Setting

Heuristics Chosen. For the purpose of this paper, we focused on three VVO heuristics only, as we would just like to show here 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 $\langle n, k, b \rangle$ up to a given n); again, given more time, instances generated in a more fine-grained way could be used instead and help make our results more precise. Finally, we calculated the median cost of only 5 instances for each $\langle n, k, b \rangle$ family, in order to offset the impact of exceptionally hard instances and failed searches; given more time, many more instances should be generated for each $\langle n, k, b \rangle$. We used the following three VVO heuristics:

– The *default VVO heuristic*, here named H_1, 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*, here named H_2, 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 (for *true*) first (as we need only consider the Boolean domain).
– The *dynamic VVO heuristic*, here named H_3, says that the next variable is chosen in a way that maximises the sum of the promises [7] of its values, and that it is labelled with the minimum promising value.

The default VVO heuristic does not introduce any overhead. The static one has a pre-processing overhead, while the dynamic one is the most costly one, as it incorporates calculations at each labelling step. We tested the effect of these heuristics by fixing the propagation and search algorithms, namely by using the ones of SICSTUS CLP(FD) [2].

Instance Characterisation. The finite-domain constraint store for *any* subset problem is over a set of Boolean variables and contains an instance-dependent number of binary constraints as well as a summation constraint. For binary CSPs, a family of instances is usually characterised by a tuple $\langle n, m, p_1, p_2 \rangle$ [17], where n is the number of variables, m is the (assumed constant) domain size for all variables, p_1 is the (assumed constant) constraint density, and p_2 is the (assumed constant) tightness of the individual constraints.

In our experiments, the variable count n is the number of Boolean variables; we varied it over the interval 10..200, by increments of 10. The domain size m is fixed to 2 as we only consider the Boolean domain $\{0, 1\}$ in subset problems, so that m can be discarded. The constraint density p_1 is $\frac{b}{n(n-1)/2}$, where b is the number of actually posted binary constraints; rather than varying b (as initially advocated), we varied the values of p_1, using the interval 0.1..1, by increments of 0.1, as this also leads to an interval of b values. Since the considered binary constraints are of the form $\neg(B_i \wedge B_j)$, their tightness is always equal to $\frac{3}{4}$ and need thus not be varied. The tightness of the summation constraint however varies, as it is $\binom{n}{k}/2^n$, where k is the desired size of the subset; instead of varying the values of p_2, we varied (as initially advocated) the values of k, over the interval $1..n$, by increments of 1, as this also leads to an interval of p_2 values. 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 $1..n$, the corresponding values of p_2 do not exhibit a constant increment within 0..1. Hence we used $\langle n, p_1, k \rangle$ to characterise instance families.

Experiments. Having thus chosen the intervals and increments for the parameters describing the characteristics of families of instances of subset problems, we randomly generated many different instances and then used the three chosen heuristics in order to find the first solution or prove that there is no solution. Some of the instances were obviously too difficult to solve or disprove 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" available heuristic for a given instance family when all three heuristics timed out on the 5 instances we generated.

The obtained results are reported, in Table 1, as $\langle n, p_1, k, c_1, c_2, c_3 \rangle$ tuples, where c_i is (here) the median CPU-time for heuristic i. The scale of the timings, as well as the used hardware and software platforms are irrelevant, as we are only interested in the relative behaviour of the heuristics. We can see that indeed no heuristic outperforms all other heuristics, or is outperformed by all the others, or never outperforms all the others. Moreover, the collected costs look very

n	p_1	k	c_1	c_2	c_3
⋮	⋮	⋮	⋮	⋮	⋮
100	0.2	6	40	970	2030
⋮	⋮	⋮	⋮	⋮	⋮
110	0.2	22	time-out	20	1880
⋮	⋮	⋮	⋮	⋮	⋮
130	0.3	18	time-out	10250	5150
⋮	⋮	⋮	⋮	⋮	⋮

Table 1. Tabulated results of the experiments

unpredictable and have many outlyers. This confirms Minton's and Tsang *et al.*'s results, and also shows that human intuition may break down here (especially when dealing with blackbox solvers).

In order to analyse the effects of each heuristic on different instances, we drew various charts from Table 1, for example by keeping n and p_1 constant and plotting the costs for each k. Figure 2 shows an example of the CPU-time behaviours of the three heuristics on the instances where $n = 110$ and $p_1 = 0.4$.

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

– As k gets smaller, for a given p_1 and n, the default VVO heuristic almost always outperforms the others.
– As k gets larger, for a given p_1 and n, the performance of the default VVO heuristic degenerates, but the static and dynamic VVO heuristics behave much more gracefully (see Figure 2).
– Even though it is very costly to apply the dynamic VVO heuristic, it sometimes outperforms the other two heuristics.
– For some of the instances, all the heuristics failed to find a solution, or prove the non-existence of solutions, within a reasonable amount of time.

3.3 The Meta-Heuristic

Designing the Meta-Heuristic. Using the obtained table as a lookup table, it is straightforward to devise a (non-adaptive) 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 heuristic to actually run on this instance. Considering the simplicity of these measures, the (constant) runtime overhead is negligible, especially that it nearly always pays off anyway. The meta-heuristic thus gives rise to an instance-independent program that is guaranteed to run, for *any* instance, (almost exactly) as fast as the fastest considered heuristic for its instance family.

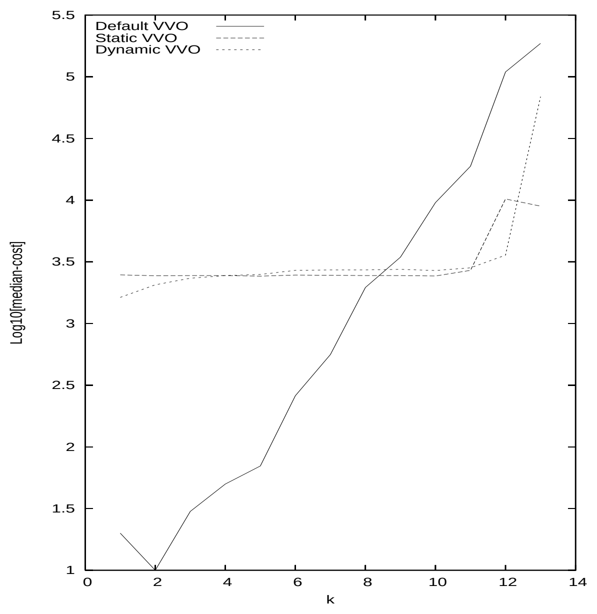

Fig. 2. Median cost in terms of k for the three heuristics on $n = 110$ and $p_1 = 0.4$

Improving the Meta-Heuristic. A very important observation is that, for many $\langle n, p_1, k\rangle$ families, *all* instances can be shown to have no solution, so that the best heuristic is to fail immediately, and there is no need to even choose between the actual heuristics in the lookup table. This is here the case when the following holds:

$$\binom{n}{2} - b \;<\; \binom{k}{2}$$

where n is the number of Boolean variables, b is the number of binary constraints of the form $\neg(B_i \wedge B_j)$, and k is the size of the desired subset. Note that b also is the number of 2-combinations of Boolean variables that cannot simultaneously be 1 (which here stands for *true*), so $\binom{n}{2} - b$ is the number of 2-combinations of Boolean variables that can simultaneously be 1. As k also is the number of Boolean variables that must simultaneously be 1, we have that $\binom{k}{2}$ combinations of Boolean variables must be simultaneously 1. Therefore, if $\binom{n}{2} - b$ is strictly less than $\binom{k}{2}$, then no solution exists.

This can be exploited by overwriting some entries in the lookup table, or, better, by reducing the number of experiments and then adding the corresponding $fail$ entries to the lookup table. This leads to our meta-heuristic being sometimes strictly (and possibly significantly) *faster* than *all* the underlying heuristics, if not faster than *any* other heuristic! This only became possible because we (need to) detect the family to which the current instance belongs.

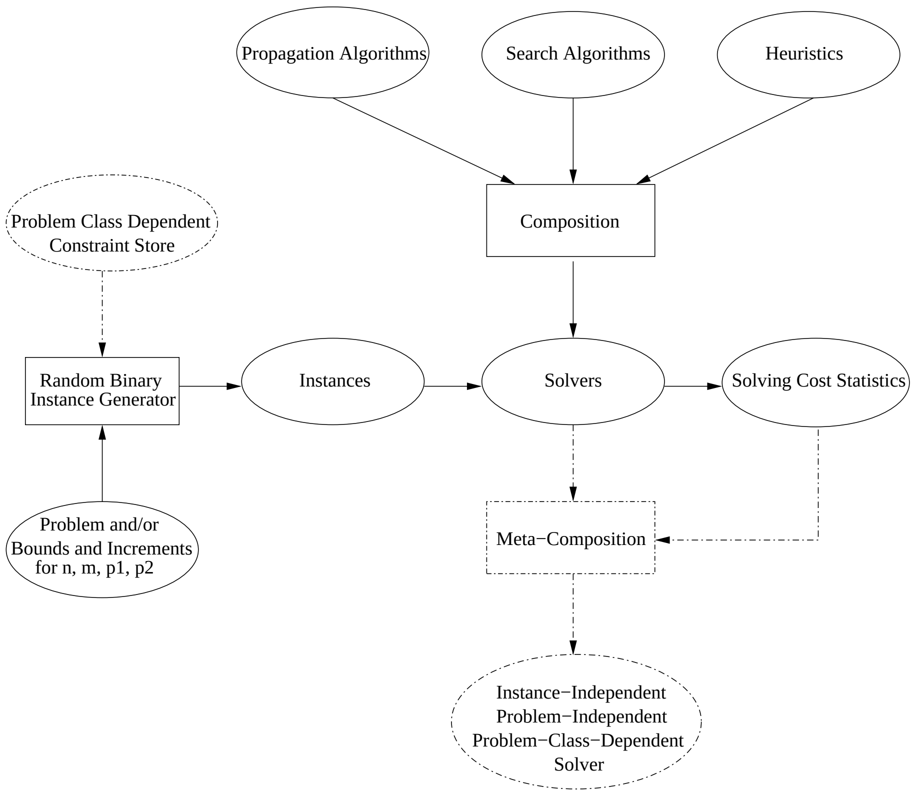
Propagation Algorithms
Search Algorithms
Heuristics
Problem Class Dependent Constraint Store
Composition
Random Binary Instance Generator
Instances
Solvers
Solving Cost Statistics
Problem and/or Bounds and Increments for n, m, p1, p2
Meta−Composition
Instance−Independent Problem−Independent Problem−Class−Dependent Solver

Fig. 3. Proposed extension to the classical quest for solvers

We have furthermore made a regression analysis to derive an evaluation function, instead of using the full look-up table. This does not speed up the resulting programs, but the size of the solver shrinks dramatically, as the look-up table does not have to be incorporated.

Methodological Contribution. The classical approach (shown in full lines in Figure 3) to detecting good solvers starts by composing several solvers from available propagation algorithms, search algorithms, and heuristics. Random instances of binary CSPs are then generated for a given problem and/or given bounds and increments for the $\langle n, m, p_1, p_2 \rangle$ parameters that govern binary CSPs. Running the composed solvers for the generated instances yields solving-cost statistics, which are always evaluated manually.

Our approach (shown in dashed lines in Figure 3) extends this scenario by making the obtained statistics an input to a new process, namely meta-composition of solvers, so as to build a problem-and-instance-independent but problem-class-dependent solver that is guaranteed to outperform all the other ones. Also, instances are here just generated for the considered problem-class-specific CSP (which is not necessarily a binary CSP).

4 Conclusion

4.1 Summary

We have shown how to map an entire class of NP-complete CSPs to a generic constraint store, and we have devised a class-specific but problem-independent meta-heuristic that chooses a suitable instance-specific heuristic. This work is thus a continuation of Tsang *et al.*'s research [17] on mapping heuristics to application domains, and an incorporation of Minton's and Tsang *et al.*'s findings about the sensitivity of heuristics to (distributions of) instances. The key insight is that we can 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 brute-force-designed and simple meta-heuristic, *can* be devised for an entire problem class, without regard to its problems or their instances. Our restriction to the (NP-complete) class of subset problems where g only constrains the size of the subset and p is not *true* was just made to simplify our presentation, as we only aimed at proving the *existence* of meta-heuristics for (useful) problem classes.

Considering the availability of such a meta-heuristic, programmers can be encouraged to model their CSPs as subset problems rather than in a different way (if this 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 issues can thus be taken care of by the solver, leaving only the declarative issue of modelling the CSP to the programmers, thus extending the range and size of CSPs that they can handle efficiently. Further advances along these lines will bring us another step closer to the holy grail of programming (for CSPs).

4.2 Related Work

This work follows the call of Tsang *et al.* for mapping combinations of solver components to application domains [17]. 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 solver otherwise constant.

Also closely related to our work is Minton's MULTI-TAC system [14], which automatically synthesises an instance-distribution-specific solver, given a high-level model of some CSP and a set of training instances (or a generator thereof). His motivation also was that heuristics depend on the distribution of instances. However, we differ from his approach in various ways:

- While good performance of a solver synthesised by MULTI-TAC is only guaranteed for the actual distribution of the *given* training instances, we advocate the off-line brute-force approach of generating *all* possible instance families for a given problem class and analysing their run-time behaviours towards the identification of a suitable meta-heuristic that is guaranteed to choose the “best” available heuristic for *any* considered instance.
- While MULTI-TAC uses a synthesis-time brute-force approach to generate candidate problem-specific heuristics, we only choose our heuristics from (variations of) 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 kind of meta-heuristic can be pre-computed once and for all, in a problem-and-instance-independent way for an entire class of problems, and the user thus need *not* provide more than a high-level problem model.
- While MULTI-TAC features very long synthesis/compilation times for each problem, our approach is to eliminate them by pre-computing the results for entire problem classes.

A similar comparison can be made with Ellman's DA-MSA system [3].

Our work differs from the problem complexity (as opposed to algorithm complexity) work of Williams and Hogg [19] as follows. Whereas they propose an *analytical* approach of charting the *search space*, under any search algorithm, towards *predicting* the location of hard instances and the fluctuations in solving cost, we propose an analytical approach of first charting the *constraint store* and then actually *determining*, with an *empirical* approach, the same things, also under any search algorithm. Furthermore, their kind of analysis has to be repeated for every *problem*, while our approach can be deployed onto an entire problem *class*.

4.3 Future Work

As our approach rests on the assumption that all instances of an $\langle n, p_1, k\rangle$ (or, equivalently, $\langle n, k, b\rangle$) family will benefit from the same heuristic, namely the one chosen for the instance that had the median cost, more analytical and empirical work is needed to better understand and model the variance in behaviour inside a family, and to understand whether $\langle n, k, b\rangle$ is an effective characterisation of subset problem instances.

We currently investigate the design of *adaptive* meta-heuristics [1, 9] that choose a (possibly different) heuristic after each labelling iteration, based on the current sub-problem, rather than sticking to the same initially chosen heuristic all the way. The hope is that the performance would increase even more. In [12], we explain why the heuristic H_2 of this paper often outperforms all the other considered ones (and many others) when there *is* a solution. This allowed us to design, in [13], a first adaptive meta-heuristic for subset problems, and we now try to integrate it with the (non-adaptive) meta-heuristic ideas of this paper.

We should also produce solving-cost statistics in a more finegrained way (with *more* than 5 instances of *more* $\langle n, p_1, k \rangle$ families until some realistic upper bound for n) and involve more known heuristics, so as to finetune the lookup-table for our meta-heuristic. Our principle of joining heuristics into a meta-heuristic, for a given problem class, can be generalised to *all* solver components, leading to a joining of entire solvers into a meta-solver, for a given problem class (as already shown in Figure 3). All this is just a matter of having the (CPU) time to do so.

Other meta-heuristics for different classes of subset problems will be devised, for cases where g has other constraints than the size of the subset. The studied class of subset problems can be generalised into the class of s-subset problems (where a maximum of s subsets of a given set have to be found, subject to some constraints) [10]. Another extension is the coverage of (s-)subset *optimisation* problems, instead of just the decision problems studied here.

Finally, we are planning to investigate other classes of problems, namely *mapping problems* (where a mapping between two given sets has to be found, subject to some constraints) [4], *permutation problems* (where a sequence representing a permutation of a given set has to be found, subject to some constraints) [4], and *sequencing problems* (where sequences of bounded size over the elements of a given set have to be found, subject to some constraints) [6], or any combinations thereof [6], in order to derive further meta-heuristics. These will be built into the compiler of our ESRA constraint modelling language [6], which is more expressive than even OPL [18]. This will help us fulfill our design objective of also making ESRA more declarative than OPL, namely by allowing the omission of a VVO heuristic, without compromising on efficiency compared to reasonably competent programmers.

Acknowledgements

This research is partly funded under grant number 221-99-369 of TFR (the Swedish Research Council for Engineering Sciences). Many thanks to our colleague Dag Sörbom for his help with the regression analysis, and to Edward Tsang (University of Essex, UK) for his invaluable comments. We are also grateful to the referees for their useful suggestions (if not misunderstandings).

References

1. J.E. Borrett, E.P.K. Tsang, and N.R. Walsh. Adaptive constraint satisfaction: The quickest-first principle. In *Proc. of ECAI'96*, pp. 160–164. John Wiley & Sons, 1996.
2. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In: H. Glaser, P. Hartel, and H. Kuchen (eds), *Proc. of PLILP'97*, pp. 191–206. LNCS 1292. Springer-Verlag, 1997.
3. T. Ellman, J. Keane, A. Banerjee, and G. Armhold. A transformation system for interactive reformulation of design optimization strategies. *Research in Engineering Design* 10(1):30–61, 1998.
4. 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.

5. 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.
6. P. Flener, B. Hnich, and Z. Kızıltan. Compiling high-level type constructors in constraint programming. In: I.V. Ramakrishnan (ed), *Proc. of PADL'01*. LNCS, this volume. Springer-Verlag, 2001.
7. P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proc. of ECAI'92*, pp. 31–35. John Wiley & Sons, 1992.
8. C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints* 1(3):191–244, 1997.
9. J.M. Gratch and S.A. Chien. Adaptive problem-solving for large scale scheduling problems: A case study. *J. of Artificial Intelligence Research* 4:365–396, 1996.
10. B. Hnich and Z. Kızıltan. Generating programs for k-subset problems. In P. Alexander (ed), *Proc. of the ASE'99 Doctoral Symposium*. 1999.
11. B. Hnich, Z. Kiziltan, and P. Flener. A meta-heuristic for subset decision problems. In: K.R. Apt, E. Monfroy, and F. Rossi (eds), *Proc. of the 2000 ERCIM/CompuLog Workshop on Constraint Programming*. 2000.
12. Z. Kızıltan, P. Flener, and B. Hnich. A labelling heuristic for subset problems. Submitted for review. Available via http://www.dis.uu.se/~pierref/astra/.
13. Z. Kızıltan and P. Flener. An adaptive meta-heuristic for subset problems. Submitted for review. Available via http://www.dis.uu.se/~pierref/astra/.
14. S. Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints* 1(1–2):7–43, 1996.
15. T. Müller. Solving set partitioning problems with constraint programming. In *Proc. of PAPPACT'98*, pp. 313–332. The Practical Application Company, 1998.
16. E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
17. E.P.K. Tsang, J.E. Borrett, 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.
18. P. Van Hentenryck. *The* OPL *Optimization Programming Language*. The MIT Press, 1999.
19. C.P. Williams and T. Hogg. Exploiting the deep structure of constraint problems. *Artificial Intelligence* 70:73–117, 1994.