# Towards Inferring Labelling Heuristics for CSP Application Domains

Zeynep Kızıltan, Pierre Flener, and Brahim Hnich

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

**Abstract.** Many real-life problems can be represented as constraint satisfaction problems (CSPs) and then be solved using constraint solvers, in which labelling heuristics are used to fine-tune the performance of the underlying search algorithm. However, few guidelines have been proposed for the application domains of these heuristics. If a mapping between application domains and heuristics is known to the solver, then modellers can — if they wish so — be relieved from figuring out which heuristic to indicate or implement. Instead of inferring the application domains of (known) heuristics, we advocate inferring (known or new) heuristics for application domains. Our approach is to first formalise a CSP application domain as a family of models, so as to exhibit the generic constraint store for all models in that family. Second, family-specific labelling heuristics are inferred by analysing the interaction of a given search algorithm with this generic constraint store. We illustrate our approach on a domain of subset problems.

## 1 Introduction

Many real-life problems are *constraint satisfaction problems* (CSPs), where appropriate values for the variables of the problem have to be found within their domains, subject to some constraints. Examples are production planning subject to demand and resource availability, air traffic control subject to safety protocols, etc. Many of these problems can be programmed as constraint models and then be solved using constraint solvers, such as CLP(FD) [2] and OPL [17].

*Constraint solvers* are equipped with a *search algorithm*, such as forward-checking, and *labelling heuristics*, one of which is the default. To enhance the performance of constraint models, a lot of research has been made in recent years to develop new labelling heuristics, which concern the choice of the next variable to branch on during the search and the choice of the value to be assigned to that variable. These heuristics significantly reduce the search space [15].

However, little is said about the application domains of these heuristics, so modellers find it difficult to decide when to apply a particular heuristic, and when not. Indeed, there is no universally best heuristic for all instances of all constraint models (see, e.g., [16]), unless NP=P. Thus, we are only told that a particular heuristic was "best" for the particular instances used to carry out

some experiments with some particular models. Therefore, the performance of heuristics is not only model-dependent but also instance-dependent, i.e., for a given constraint model, a heuristic can perform well for some (distributions on the) instances, but very poorly on others; this is taken into account by some generators of model-specific solvers [3, 9, 12].

Instead of inferring the application domains of (known) heuristics, we advocate inferring (known or new) heuristics for application domains. Obviously, the "smaller" an application domain, the "better" its inferrable heuristics. Our two-step approach is to first formalise an application domain as a family of CSP models, so as to exhibit the generic constraint store for all models in that family. Second, the interaction — for a given search algorithm — between the constraints in this generic store and the domain propagation during search is examined, so as to infer suitable heuristics for any model in that family. Due to the instance sensitivity of heuristics, the outcome of this process usually is a *set* of heuristics, rather than a single one. In this paper, we illustrate this approach on a domain of subset problems.

If a mapping between application domains and heuristics is known to the solver, then modellers can — if they wish so — be relieved from the procedural aspect of modelling, namely figuring out which heuristic to indicate or implement. Forcing modellers to deal with this procedural aspect may not only add a challenging step but also has the disadvantage that they must commit — at modelling time — to a *single* heuristic and thus expose their models to the instance sensitivity of heuristics. In companion work [7, 11], we address the issue of selecting or switching — at solving time — among the inferred family-specific heuristics resulting from our approach, according to the instance to be solved. Our ultimate aim is thus a new generation of more intelligent solvers that allow CSP modellers to concentrate on the declarative aspect of modelling, without compromising (much) on efficiency.

This paper is organised as follows. In Section 2, we introduce the notion of family of CSP models as a formalisation of an application domain. We illustrate this with a domain of subset problems and exhibit a generic finite-domain constraint store of a family for this domain. Then, in Section 3, we present our analysis of this generic constraint store, infer two labelling heuristics, and show our initial empirical results. Finally, in Section 4, we conclude, compare with related work, and discuss directions for future research.

## 2 CSP Model Families

Informally, an *application domain* is a set of "related" CSPs. For instance, in the *SUBSET* domain, a given number of elements have to be selected from a given finite set such that any two of them satisfy some constraint $p$. In this domain, CSPs are related in the sense that the actual constraint $p$ differs between them. Sample CSPs in this domain are finding a clique of a given size within a given graph (where $p$ requires that any two vertices of the clique be connected by an edge of the graph) and finding an independent subset of a given size among

the vertices of a given graph (where $p$ says that any two vertices of the subet must not be connected by an edge of the graph). Application domains of coarser granularity are scheduling, configuration, resource allocation, and so on.

For a given constraint modelling language, a *CSP model family* is an open CSP model in that language, '*open*' in the sense that some of its (predicate or type) symbols are neither primitive to the language nor defined in the model. An actual *CSP model* is *closed*, in the sense that all its symbols must be primitive or defined. From a model family, a model can thus be obtained by substituting closed types and closed formulas for all its open symbols, and possibly by adding parameters. Model families can be used to formalise application domains. There are in general several ways of formalising a domain as a model family, in a given language, namely depending on the chosen data modelling. An *instance* of a CSP model $M$ is obtained from $M$ by replacing all its formal input parameters by actual values and dropping the universal quantifications on these parameters. An instance of a model is thus also a model, albeit without input parameters.

*Example 2.1.* Assume CSP models are written in a very expressive, purely declarative, typed, set-oriented, first-order logic constraint modelling language, such as our ESRA [6, 4], which is designed to be higher-level than even OPL [17]. (We can automatically compile [5] ESRA programs into lower-level languages such as OPL.) Since ESRA has set variables (unlike OPL), the following (sugared version of an) ESRA model family is a candidate formalisation of the *SUBSET* domain:

$$\begin{array}{c} \forall T, S : set(\alpha)\,.\,\forall k : int\,.\,(subset(T^+, k^+, S) \leftrightarrow S \subseteq T \wedge size(S, k) \wedge \\ \forall t_i, t_j : \alpha\,.\,(t_i \in S \wedge t_j \in S \wedge t_i \neq t_j \rightarrow p(t_i, t_j))) \end{array} \qquad (Subset)$$

where the superscript $^+$ designates the input parameters. In words, sets $S$ and $T$ of elements of type $\alpha$ are in the $subset/3$ relation with integer $k$ iff $S$ is a set of $k$ elements from $T$, such that any two distinct elements $t_i$ and $t_j$ of $S$ satisfy constraint $p$. The only open symbols are type $\alpha$ and constraint $p$, as $size$, $\subseteq$, $\in$, and $\neq$ are primitives of ESRA, with the usual meanings. From the $Subset$ model family, we can obtain the following (sugared) ESRA model:

$$\begin{array}{c} \forall V, C : set(int)\,.\,\forall k : int\,.\,\forall E : set(int \times int)\,. \\ (clique_k(\langle V^+, E^+\rangle, k^+, C) \leftrightarrow C \subseteq V \wedge size(C, k) \wedge \\ \forall v_i, v_j : int\,.\,(v_i \in C \wedge v_j \in C \wedge v_i \neq v_j \rightarrow \langle v_i, v_j\rangle \in E)) \end{array} \qquad (clique_k)$$

It is a model for finding a clique $C$ of an undirected graph (given through its integer vertex set $V$ and edge set $E$), such that the clique has $k$ vertices.

*Example 2.2.* At a lower level of expressiveness, say when set variables are not available (such as in CLP(FD) [2] and OPL [17]), the usual representation of an unknown subset $S$ of a given *finite* set $T$ (of $n$ elements) is a mapping from $T$ into Boolean variables (in $\{0, 1\}$), that is one conceptually maintains $n$ couples $\langle t_i, B_i\rangle$ where the (initially non-ground) Boolean $B_i$ expresses whether the (always ground) element $t_i$ of $T$ is a member of $S$ or not:[1]

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

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

This Boolean representation of set variables consumes more memory than the set-interval representation of CONJUNTO [8] and OZ [13], but both have been shown to create the same $O(2^n)$ search space [8].

Given this Boolean representation of the sought subset $S$, restricting its size to $k$ can be expressed as the following $n$-ary constraint:

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

Let us also look at the remaining part of *SUBSET*, which requires that any two distinct elements of the subset $S$ of $T$ must satisfy a constraint $p$. Formally (using the sugared ESRA syntax again, for the sake of symbolic reasoning):

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

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

$$\forall t_i, t_j : \alpha \,.\, t_i \in T \wedge t_j \in T \wedge t_i \neq t_j \wedge \neg p(t_i, t_j) \rightarrow \neg(B_i \wedge B_j) \tag{3}$$

The sugared version of an OPL/CLP(FD) model family formalising the *SUBSET* domain thus consists of constraints (2) and (3); we denote it by $Subset_B$. For any two distinct elements $t_i$ and $t_j$ of the given set $T$, with Boolean variables $B_i$ and $B_j$, if $p(t_i, t_j)$ does not hold, the following binary constraint arises:

$$\neg(B_i \wedge B_j) \tag{4}$$

It is crucial to note that the actual finite-domain constraints are thus *not* in terms of $p$, hence $p$ can be *any* formula. Therefore, the generic finite-domain constraint store for *any* instance of *any* model of the $Subset_B$ family is over a set of (only) Boolean variables. It contains an instance-dependent number of binary constraints of the form (4), as well as the (always unique) $n$-ary constraint (2).

As the set-interval representation of set variables does not allow the definition of some (to us) desirable high-level primitives, such as universal quantification over elements of non-ground sets, the set variables of ESRA (see Example 2.1) are compiled [5, 6] using the Boolean representation of Example 2.2. In the remainder of this paper, our approach to inferring labelling heuristics from an application domain is illustrated on the *SUBSET* domain, and we (thus) focus on its Boolean modelling in the $Subset_B$ family.

## 3 Inferring Labelling Heuristics

It is known that the order in which the variables are considered for instantiation, and the order in which the values are attempted for assignment to variables during search have a substantial impact on the number of backtracks performed and the time taken by a search algorithm to solve a CSP model. Deciding on these orders is the objective of labelling heuristics. We now infer some labelling heuristics for the *SUBSET* domain by examining the domain propagation performed on the generic constraint store — for the $Subset_B$ family — by a search algorithm during labelling. For the sake of illustration, we here choose the forward checking (FC) algorithm, which is used in many solvers. It works as follows: Whenever a variable is labelled by a value $v$, the values of the future variables that are inconsistent with $v$ are removed from the domains of these variables.

In Section 3.1, we present our analysis of the obtained generic constraint store. Next, in Section 3.2, we infer some FC labelling heuristics for $Subset_B$ models. Finally, in Section 3.3, we report on our initial experimental results.

### 3.1 Analysis of the Generic Constraint Store

We analyse the generic constraint store using the values $n$ (the size of the given set $T$, hence the number of Boolean variables involved) and $k$ (the given size of the sought subset $S$). In models of the $Subset_B$ family, each Boolean variable $B_i$ in $\{B_1, \ldots, B_n\}$ is at any moment associated with the set $V_i$ of still unassigned variables $B_j$ (where $1 \leq j \leq n$) that constrain $B_i$ with a binary constraint of the form (4). A binary constraint of this form requires that the variables $B_i$ and $B_j$ cannot simultaneously be assigned 1. Furthermore, the $n$-ary constraint (2) restricts all the variables such that $k$ of them must be assigned 1. Let $k_0$ (resp. $k_1$) be the *current* number of variables that have yet to be assigned 0 (resp. 1). Initially (before the labelling), $k_0 = n-k$ and $k_1 = k$. During labelling, the values of $k_0$ and $k_1$ decrease because of the assignments and propagation. If either $k_0$ or $k_1$ reaches 0, the propagation caused by the $n$-ary constraint forces the other one to also become 0. Therefore, at the end (after the labelling), $k_0 = k_1 = 0$. Note that the mathematical variables $V_1 \ldots V_n$, $k_0$, $k_1$ are only explanatory devices, but not actually stored and manipulated anywhere.

We now monitor the FC propagations triggered by the assignment of values (from $\{0, 1\}$) to the Boolean variables. The ordering of the variables and values is irrelevant in this analysis: suitable labelling heuristics will be inferred in Section 3.2. When $k_0 > 0$ and $k_1 > 0$, we consider two cases, namely Case A, the assignment of 0, and Case B, the assignment of 1 to the chosen variable, say $B_i$.

**Case A.** If $B_i$ is assigned 0, the current number of variables that have yet to be assigned 0 is decremented by 1, so $k_0$ becomes $k_0 - 1$. Two sub-cases arise now:

- If $k_0 = 0$ now, then all the $k_1$ yet unassigned variables are assigned 1 during propagation due to (only) the $n$-ary constraint (2), leading to $k_1 = 0$ also. Now exactly $n - k$ variables have been assigned 0 and $k$ variables have been assigned 1. However, if there is a binary constraint of the form (4) between

any two of these $k_1$ variables, then this assignment fails, which leads to backtracking. Otherwise, this assignment succeeds.
- If $k_0 > 0$ still, then, for all $v \in V_i$, the domain of $v$ remains the same, because the assignment of 0 to any variable in a binary constraint of the form (4) always succeeds without propagation.

By the instantiation of a variable by 0, there is thus a possibility of backtracking only if $k_0$ reaches 0, because the assignment may fail.

**Case B.** If $B_i$ is assigned 1, the current number of variables that have yet to be assigned 1 is decremented by 1, so $k_1$ becomes $k_1 - 1$. Two sub-cases arise now:

- If $k_1 = 0$ now, then all the $k_0$ yet unassigned variables are assigned 0 during propagation due to (only) the $n$-ary constraint (2), leading to $k_0 = 0$ also. Now exactly $k$ variables have been assigned 1 and $n - k$ variables have been assigned 0, without violating any constraints. Indeed, as seen in Case A, the assignment of 0 to a variable fails only if $k_0$ becomes 0 *and* there is a binary constraint between any two of the $k_1$ variables. However, there are here *no* unassigned variables left, as $k_1 = 0$ already. Therefore, this assignment always succeeds.
- If $k_1 > 0$ still, then, for all $v \in V_i$, the variable $v$ is assigned 0 during propagation because of the binary constraints of the form (4). Thus, $k_0$ becomes $k_0 - |V_i|$. The new value of $k_0$ now gives rise to the following sub-sub-case analysis:
  - If $k_0 < 0$ now, then one of these assignments must fail and immediate backtracking occurs.
  - If $k_0 = 0$ now, then all the $k_1$ yet unassigned variables are assigned 1 during propagation, leading to $k_1 = 0$ also. As seen in Case A, if there is a binary constraint of the form (4) between any two of these $k_1$ variables, then this assignment fails, which leads to backtracking. Otherwise, this assignment succeeds.
  - If $k_0 > 0$ now, then this assignment succeeds.

By the instantiation of a variable by 1, there is thus a possibility of backtracking only if $k_0$ reaches 0 first. Should $k_0$ become negative, the assignment fails, and thus an immediate backtracking occurs. On the other hand, the assignment always succeeds if $k_1$ reaches 0 first.

It is *very* important to notice that Case B may include Case A. On the other hand, Case A never includes the general situation of Case B. Therefore, the analysis became of *finite* size and complete, as there is no case where it is impossible to exactly foretell *all* propagations!

### 3.2 Inference of Heuristics

In models of the $Subset_B$ family, the assignment — under FC search — of 0 to a Boolean variable triggers propagation *only* when $k_0$ reaches 0, and this *independently* of the order of the variables being instantiated by 0 so far. Therefore, if the *set* of variables that will be assigned 0 is not chosen carefully (e.g., when

there are no binary constraints between them, in which case there probably are binary constraints between the other variables), backtracking is unavoidable once $k_0$ reaches 0. The only way to avoid backtracking is to choose the right set of $n-k$ variables that are assigned 0. However, finding such a subset of the Boolean variables is itself a subset problem.

The assignment of 1 to a variable is noteworthy because *every* assignment caused by propagation upon $k_1 = 0$ succeeds, so that no backtracking can happen. Also, the order of the variables being assigned 1 is quite important because it can significantly affect the decrease in $k_0$. Indeed, as seen in Case B, if $k_1 > 0$ still, then $k_0$ becomes $k_0 - |V_i|$. The variable $B_i$ being assigned 1 is associated with a set $V_i$ (the set of the still unassigned variables that constrain $B_i$) that thus directly affects the decrement in $k_0$. If the variables being assigned 1 are ordered in a way that they do not cause much decrease in $k_0$, then backtracking when $k_0 < 0$ and any possible backtracking when $k_0 = 0$ are delayed. Backtrack-free assignment is thus guaranteed by allowing $k_1$ to reach 0 first. However, backtrack-free assignment is not guaranteed if it is $k_0$ that reaches 0 first.

We can thus infer the following two labelling heuristics from the previous considerations:

- *If there is at least one solution*, we should instantiate some variables by 1, and try to keep each $|V_i|$ as small as possible if we want $k_1$ to reach 0 first (which leads to backtrack-free assignment). Thus, during FC search, if we choose a variable that is participating in the *smallest* number of binary constraints, then we force $k_1$ to become 0 before (or at the same time) as $k_0$ does, because, by this way, we achieve a small decrease in $k_0$. This heuristic can be seen as an instance of the succeed-first principle.
- *If there is no solution*, then it is impossible to reach the state $k_1 = 0$. Search effort can then be saved by forcing the search to reach a state with definite backtracking (when $k_0 < 0$) or possible backtracking (when $k_0 = 0$) as soon as possible. Thus, during FC search, if we choose a variable that is participating in the *largest* number of binary constraints, then we force $k_0$ to be negative or to become 0 before $k_1$ does, because, by this way, we achieve a big decrease in $k_0$. The value ordering is thus irrelevant. This heuristic can be seen as an instance of the fail-first principle.

As it is initially unknown whether there is a solution or not, it is very difficult to choose which of these two heuristics to use in order to guide the search process. This paper is only concerned with the inference of heuristics; the issue of deciding when to use which one, or when to switch between them, is addressed in companion work [7, 11].

Following these considerations, we implemented the following static labelling heuristics, namely in SICSTUS CLP(FD) (which has an FC solver):

- $H_s^1$, which chooses the variable that is constraining the smallest number of variables, and assigns the value 1 first.
- $H_l^0$ (resp. $H_l^1$), which chooses the variable that is constraining the largest number of variables, and assigns the value 0 (resp. 1) first.

Being static, these labelling heuristics choose a variable that is *initially* constraining the smallest/largest number of variables. Note that this implementation of the heuristics is our choice, but that the heuristics could be implemented in another way, say by re-ordering the variables at solving-time. Investigation of the superiority or the inferiority of such dynamic variable orderings, which choose a variable that is constraining the smallest/largest number of the future (yet unassigned) variables, to the static ones is left as future work.

### 3.3 Experiments with the Heuristics

**Experimental Setting.** We measured the cost (in CPU time and in number of backtracks) of our heuristics on a very large number of instances of the models of the $Subset_B$ family. These experiments confirmed the anticipated strengths and weaknesses of the heuristics, which are exploited in our companion work on deciding when to use which heuristic, or when to switch between them [7, 11].

For binary CSPs, a class[2] of instances is usually characterised by a tuple $\langle n, m, p_1, p_2 \rangle$, 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. Experiments are then conducted by iterating over an interval of instance classes and generating a suitably sized sample of random instances for each class. For each sample, the median or average solving cost is computed.

However, our generic constraint store features a non-binary constraint, so we cannot literally apply this characterisation of instance classes. In any case, the latter has been criticised [1] because it is unrealistic to have a *constant* tightness $p_2$ for all constraints, so that many possible instances can never be generated. For these two reasons, we developed the following characterisation of instance classes, which is specific to the considered family. It is not subject to any of the criticisms in [1], because it exploits the structure of the generic constraint store.

The generic finite-domain constraint store for the $Subset_B$ family is parameterised by the number $n$ of Boolean variables involved (i.e., the size of the given set $T$) and the given size $k$ of the sought subset $S$, and contains an instance-dependent number $b$ of binary constraints of the form (4). The number $n$ of variables and the density $p_1$ of the constraints are kept from the previous characterisation, with $p_1$ being $\frac{b}{n(n-1)/2}$ here. The domain size $m$ is dropped, as it always is 2, because we need only consider the Boolean domain $\{0, 1\}$. Since the considered binary constraints are of the form $\neg(B_i \wedge B_j)$, their tightness always is $3/4$ and thus does not become a parameter. The tightness of the $n$-ary constraint however is $\binom{n}{k}/2^n$, and thus varies with $n$ and $k$. As we already use $n$, the size $k$ becomes the final parameter in our characterisation of instance classes, which is thus summarised by the triple $\langle n, p_1, k \rangle$.

For the purpose of this paper, we generated random instances in a coarse way, by not considering all possible values of $n$ up to a given limit. The number $n$ of variables ranged over the interval 10..120, by increments of 10. We varied

[2] A class (of instances) is not to be confused with a family (of CSP models).

the density $p_1$ over the interval 0.1..1, by increments of 0.1. The values of $k$ ranged over the interval $1..n$, by increments of 1. Considering the sizes of these intervals, the number of our experiments was huge and their execution was very time-consuming. Given more time, instances generated in a more fine-grained way could be used instead and help to make our (future) results more precise. Our objective here only is to show the heuristics in action, but not to provide the most detailed statistics for our companion work on deciding when to use which heuristic, or when to switch between them [7, 11].

Rather than only comparing the inferred heuristics to each other, we also compared them to some others. For time reasons, we restricted ourselves to the following two additional heuristics:

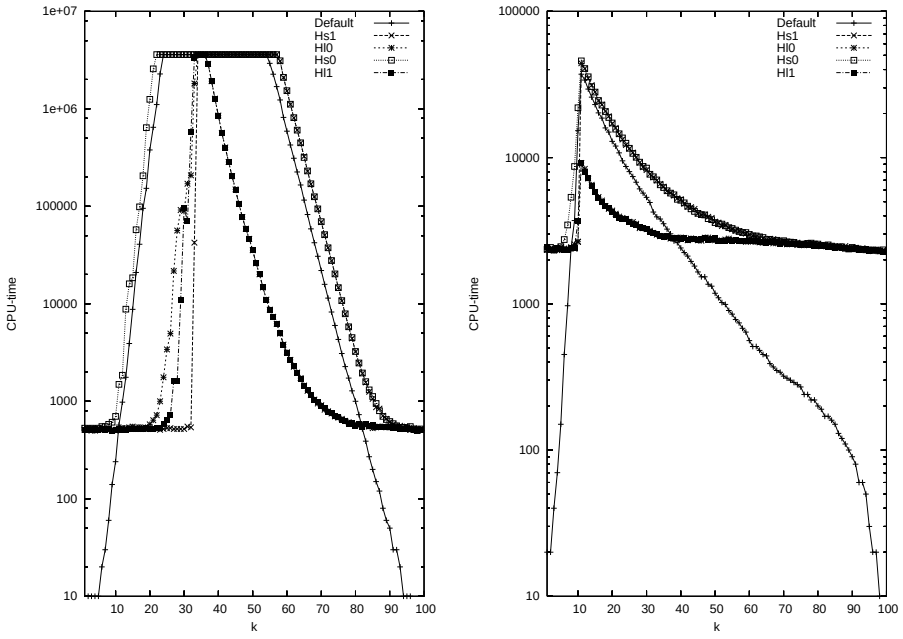
- $H_s^0$, which chooses the variable that is constraining the smallest number of variables, and assigns the value 0 first.
- $Default$, the default labelling heuristic of SICSTUS CLP(FD), which labels the leftmost variable in the provided sequence of variables, and the domain of the chosen variable is explored in ascending order (i.e., 0 first in our case).

The heuristic $H_s^0$ is a natural complement to the inferred heuristics, and was also implemented in SICSTUS CLP(FD). In the absence of a labelling heuristic provided by the modeller, each solver uses its default heuristic. Since our experiments were conducted in SICSTUS CLP(FD), its default heuristic *had to* be used here. (The experiments thus have to be repeated for each FC solver, because their default heuristics change.)

If a combination of the inferred heuristics beats — on the average over numerous instances of the family — the default heuristic of the solver, then this combination can become a family-specific and even highly instance-sensitive default heuristic of the solver. The determination of such a combination is addressed in our companion work [7, 11]. If this idea is repeated for other families, then the modellers can — if they wish so — be relieved from the procedural aspect of modelling and even be protected from the instance sensitivity of their heuristics.

Our experiments were made over *random* instances (of models) of the considered family for the following reason. Towards using *real-life* instances, we would have had to first pick some models within the considered family, but we would then have been unable to justify why these models were picked rather than some others. The purpose of our experiments [10] was to generate statistics that guide us in our companion work [7, 11], where we aim at a family-specific default heuristic for a *solver*, which must be able to handle random instances over that entire family. We do not aim at a heuristic for a specific *model*, which would have to be able to handle (only) real-life instances of (only) that model.

**Experiments.** Having thus chosen the intervals and increments for the parameters in our characterisation of an instance class, we randomly generated many different instances and then used the 5 heuristics in order to solve them or prove that they have no solutions. 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 (of 3,600,000ms) on the CPU time; upon time-out, the current number of backtracks was recorded.




(a) $p_1 = 0.1$ (b) $p_1 = 0.5$

**Fig. 1.** CPU-time (in ms) in terms of $k$ for the 5 heuristics on $n = 100$

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 median costs of the samples for each $k$. Figure 1 shows an example of the behaviours of the 5 heuristics in terms of CPU-time on the instances where $n = 100$, with $p_1 = 0.1$ and $p_1 = 0.5$, respectively. Figure 2 shows their behaviours in terms of the number of backtracks on the same instances.

These figures do not show that the generated instances exhibit three very interesting regions in terms of $k$, no matter what $n$ and $p_1$ are: up to some value $v$ of $k$, all instances have a solution; then, until some other value $w$ of $k$, some instances have a solution and some do not; beyond $w$, all instances have no solution. A visible interesting observation is that, without a time-out, the solving-times for instances increase with $k$ until some point, whereupon they decrease. With the heuristics we used, we recorded time-outs in all three of the mentioned regions. After taking the median cost of the generated sample of random instances for each class $\langle n, p_1, k\rangle$, we observed three different zones in terms of $k$: up to some value $j$ in $0..n$, the instance with the median cost has a solution; from some other value $l$ in $j+1 \,..\, n+1$, the instance with the median cost has no solution; in-between, the instance with the median cost timed out. It is in general unknown where $j$ and $l$ are compared to $v$ and $w$. The values of $j$, $l$, $v$, $w$ depend on $n$ and $p_1$.

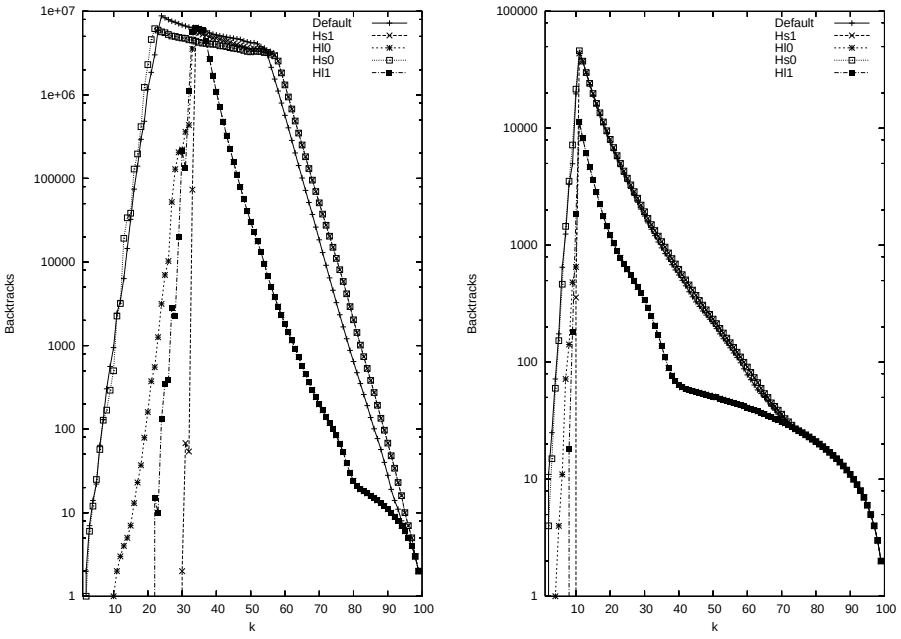
The position of $k$ relative $j$ and $l$ yields the following analysis of the behaviours of the heuristics in terms of the CPU-time they take (see Figure 1):

- Over $1..j$, the heuristic $H_s^1$ always finds a solution, in mostly *constant* CPU-time. *Default* performs the best until $k$ reaches some $d$ in $0..j$, where $d$ is small. However, over $d+1$ .. $j$, the heuristic $H_s^1$ outperforms $Default$. The heuristics $H_l^0$ and $H_l^1$ perform as well as $H_s^1$ until $k$ reaches some $i$ in $1..j$. However, over $i+1$ .. $j$, the heuristic $H_s^1$ outperforms $H_l^0$ and $H_l^1$. Heuristic $H_s^0$ usually has the worst performance. In conclusion, over $1..j$, the heuristic $H_s^1$ is the *best* over $d+1$ .. $j$, with $1..d$ being always a very small interval. The range of $k$ where $H_s^1$ performs the best varies in size with respect to $p_1$, given $n$: compare Figures 1(a) and 1(b).
- Over $j+1$ .. $l-1$, we cannot compare the heuristics because they all timed out. This can be observed in Figure 1(a) for $k$ in 34..37.
- Over $l..n$, the heuristic $H_s^1$ always proves that there is no solution, in decreasing CPU-time. Heuristic $H_s^0$ usually has the worst performance. In this range, the heuristic $H_s^1$ is always outperformed by $H_l^0$ and $H_l^1$, and performs as badly as $H_s^0$. The heuristics $H_l^0$ and $H_l^1$ perform the *best* until $k$ reaches some $i$ in $l..n$, whereupon *Default* outperforms all the others. The range of $k$ where $H_l^0$ and $H_l^1$, or *Default* perform the best varies in size with respect to $p_1$, given $n$: compare Figures 1(a) and 1(b).

The heuristic $H_s^1$ mostly performs the best when there is an observed solution. This can easily be explained by the fact that it was designed to try and find a solution, while assuming there is one. The heuristics $H_l^0$ and $H_l^1$ mostly perform the best when there is no observed solution. This is because they were designed to prove that there is no solution, while assuming there is none. The reason why $Default$ sometimes outperforms the other 4 heuristics is that it has no solving-time overhead. Somewhere in $j+1$ .. $l-1$, a phase transition from the soluble region to the non-soluble region occurs, and all the heuristics failed to efficiently handle these instances and thus timed out.

The position of $k$ relative $j$ and $l$ yields an analysis of the behaviours of the heuristics in terms of the number of backtracks they make (see Figure 2):

- Over $1..j$, the heuristic $H_s^1$ always finds a solution, mostly in 0 backtracks. *Default* always performs worse than $H_s^1$. The heuristics $H_l^0$ and $H_l^1$ initially perform as well as $H_s^1$, but start backtracking earlier. Heuristic $H_s^0$ usually has the worst performance. In conclusion, over $1..j$, the heuristic $H_s^1$ is always the *best*. The range of $k$ where $H_s^1$ performs 0 backtracks varies in size with respect to $p_1$, given $n$: compare Figures 2(a) and 2(b).
- Over $j+1$ .. $l-1$, we cannot compare the heuristics because they all timed out. This can be observed in Figure 2(a) for $k$ in 34..37.
- Over $l..n$, the heuristic $H_s^1$ always proves that there is no solution, in decreasing numbers of backtracks. Heuristic $H_s^0$ usually has the worst performance. In this range, the heuristic $H_s^1$ is always outperformed by $H_l^0$ and $H_l^1$, and performs as badly as $H_s^0$. The heuristics $H_l^0$ and $H_l^1$ perform the *best* until $k$ reaches some $i$ in $l..n$, whereupon all the 5 heuristics perform the same




(a) $p_1 = 0.1$ (b) $p_1 = 0.5$

**Fig. 2.** Number of backtracks in terms of $k$ for the 5 heuristics on $n = 100$

number of backtracks. The range of $k$ where $H_l^0$ and $H_l^1$ (resp. all the 5 heuristics) perform the best (resp. the same) varies in size with respect to $p_1$, given $n$: compare Figures 2(a) and 2(b).

The heuristic $H_s^1$ always performs the best in number of backtracks (and mostly with 0 backtracks) when there is an observed solution, because it was designed to try and find a solution, while assuming there is one. The heuristics $H_l^0$ and $H_l^1$ mostly perform the best in number of backtracks when there is no observed solution. This is because they were designed to prove that there is no solution, while assuming there is none. Somewhere in $j+1 \,..\, l-1$, a phase transition from the soluble region to the non-soluble region occurs, and all the heuristics failed to efficiently handle these instances and thus timed out.

## 4 Conclusion

Labelling heuristics may lead to a substantial reduction of the search space when solving CSP models. However, little is known about the application domains of the known heuristics. This work follows the call of Tsang *et al.* for mapping combinations of algorithms and heuristics to application domains [16]. Rather than inferring the applications domains of (known) algorithm/heuristic combinations,

we here advocate inferring (known or new) algorithm/heuristic combinations for application domains.

Our approach is to first formalise a CSP application domain as a model family, so as to exhibit the generic finite-domain constraint store for all models in that family. By analysing the interaction of an algorithm with this generic constraint store, one can then infer labelling heuristics for that family. Usually, one would at least look for a heuristic that excels at finding the first solution, one that excels at disproving the existence of solutions, and one that detects and handles the phase transition. We here illustrated this approach on a domain of subset problems, as well as on the effect of labelling heuristics for a fixed search algorithm, namely forward checking. We inferred two heuristics for this domain, one for each of the first two kinds.

We generate random instances by iterating over an interval of $\langle n, p_1, k\rangle$ instance classes and generating a suitably sized sample of random instances for each class. For each sample, if the instances are comparable (e.g., all the instances have a solution), the median cost is computed; otherwise (e.g., some instances have a solution but some do not), we cannot judge which heuristic is the "best" for this sample. We then devise a lookup table, where either the "best" heuristic for a given instance class $\langle n, p_1, k\rangle$ is designated [7], or a switching between heuristics is designated because none of the heuristics is considered to be better than another one for this class of instances [11]. This switching can be done by deploying one of the heuristics first, and monitoring the progress so as to switch to the next one in case of thrashing. This lookup table is then used by a meta-heuristic. If this meta-heuristic beats — on the average over numerous instances of the family — the default heuristic of the solver, then this meta-heuristic can become a family-specific and even highly instance-sensitive default heuristic of the solver. If this is repeated for many application domains, then modellers can — if they wish so — be relieved from indicating or implementing a heuristic at modelling-time, which often is a too early commitment anyway, due to the instance-sensitivity of heuristics.

In terms of related work, Figure 3 shows the classical approach to designing heuristics in full lines, whereas the contribution of our approach is emphasised in dashed lines and italicised text. A curved arrow from a full line to a dashed line indicates our replacement of the full line with the dashed line. We thus replace the design of a single heuristic for a CSP model in the presence of a solver (i.e., search algorithm) with the inference of a *set* of heuristics for a model-*family* by *analysis* of the propagation performed by that solver on the family-specific generic constraint store during labelling. Also, in our approach, random instances are generated *only* for the considered *family* (which does not necessarily contain binary CSPs), rather than for arbitrary (binary) CSPs.

Closely related to our work is first Minton's MULTI-TAC system [12], which automatically synthesises an instance-distribution-specific solver, given a high-level model of some CSP and a set of training instances. While MULTI-TAC uses a synthesis-time brute-force approach to generate candidate problem-specific heuristics from a set of heuristics described by a grammar, we propose inferring

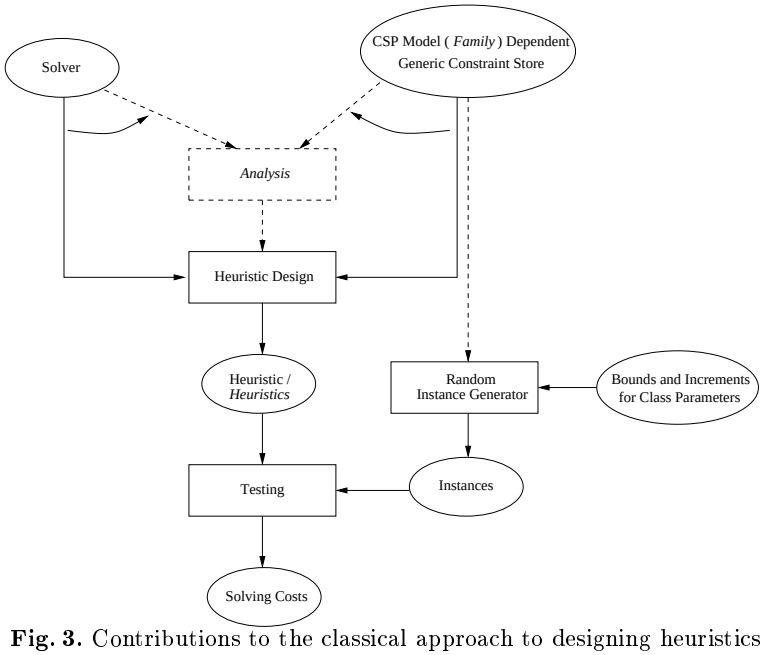



**Fig. 3.** Contributions to the classical approach to designing heuristics

candidate family-specific heuristics manually by analytically reasoning about the generic constraint store of the family. Second, Sadeh and Fox propose a probabalistic framework for the job shop scheduling domain so as to capture the search space. Based on this framework, a domain specific heuristic is derived [14]. The derived heuristic significantly reduces the search space of the instances used in the experiments. However, the instance sensitivity of heuristics is not tackled, and only one heuristic is derived for the domain.

Our future work includes investigating the superiority or the inferiority of dynamic variable orderings, which choose a variable that is constraining the smallest/largest number of the *future* (yet unassigned) variables, to the here investigated static variable orderings, which choose a variable that is *initially* constraining the smallest/largest number of variables.

We are also planning to investigate other application domains, such as *m-subset problems* (where a maximum of $m$ subsets of a given set have to be found, subject to some constraints), *relation problems* (where a relation 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) [6], 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.

All results will be built into the compiler of our ESRA constraint modelling language [6, 4], which is more expressive than even OPL [17]. This will help us

to fulfill our design objective of also making ESRA more declarative than OPL, without compromising (much) on efficiency.

**Acknowledgements**

We would like to thank Prof. Edward Tsang (University of Essex, UK) and our colleague Justin Pearson for their invaluable comments. This research is partly funded under grant number 221-99-369 of VR (the Swedish Research Council).

## References

1. D. Achlioptas, L.M. Kirousis, E. Kranakis, D. Krizanc, M.S.O. Molloy, and Y.C. Stamatiou. Random constraint satisfaction: A more accurate picture. In: G. Smolka (ed), *Proc. of CP'97*, pp. 107–120. LNCS 1330. Springer, 1997.
2. P. Codognet and D. Diaz. Compiling constraints in CLP(FD). *J. of Logic Programming* 27(3):185–226, 1996.
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. Towards relational modelling of combinatorial optimisation problems. In: Ch. Bessière (ed), *Proc. of the IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*, 2001.
5. P. Flener and B. Hnich. The syntax and semantics of ESRA. Evolving internal report of the ASTRA Team, at http://www.dis.uu.se/~pierref/astra/.
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*, pp. 229–244. LNCS 1990. Springer, 2001.
7. P. Flener, B. Hnich, and Z. Kızıltan. A meta-heuristic for subset problems. In: I.V. Ramakrishnan (ed), *Proc. of PADL'01*, pp. 274–287. LNCS 1990. Springer, 2001.
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. J.N. Hooker. Testing heuristics: We have it all wrong. *J. of Heuristics* 1:33–42, 1996.
11. Z. Kızıltan and P. Flener. An adaptive meta-heuristic for subset problems. Submitted for review. Available via http://www.dis.uu.se/~pierref/astra/.
12. S. Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints* 1(1–2):7–43, 1996.
13. T. Müller. Solving set partitioning problems with constraint programming. *Proc. of PAPPACT'98*, pp. 313–332. The Practical Application Company, 1998.
14. N.M. Sadeh and M.S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Arificial Intelligence* 86(1):1–41, 1996.
15. E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
16. 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. *Proc. of AISB'95*, pp. 203–216, 1995. IOS Press.
17. P. Van Hentenryck. *The* OPL *Optimization Programming Language*. The MIT Press, 1999.