Electronic Communications of the EASST
Volume 64 (2013)

Proceedings of the
XIII Spanish Conference on Programming
and Computer Languages
(PROLE 2013)

Improving the Search Capabilities of a CFLP(FD) System

Ignacio Castiñeiras, Fernando Sáenz-Pérez

18 pages

Improving the Search Capabilities of a CFLP(FD) System*

Ignacio Castiñeiras[1], **Fernando Sáenz-Pérez**[2]

[1] ncasti@fdi.ucm.es
Dept. Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain

[2] fernan@sip.ucm.es, http://www.fdi.ucm.es/profesor/fernan
Dept. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain

Abstract: The CFLP system $\mathcal{TOY}(\mathcal{FD})$ is implemented in SICStus Prolog, and supports $\mathcal{FD}$ constraints by interfacing the CP($\mathcal{FD}$) solvers of Gecode and ILOG Solver. In this paper, $\mathcal{TOY}(\mathcal{FD})$ is extended with new search primitives in a setting easily adaptable to other Prolog CLP or CFLP systems. The primitives are described from a solver-independent point of view, pointing out some novel concepts not directly available in the Gecode and ILOG Solver libraries. Also, we describe how to specify some search criteria at $\mathcal{TOY}(\mathcal{FD})$ level and how easily these strategies can be combined to set different search scenarios. The implementation of the primitives is described, presenting an abstract view of the requirements and how they are targeted to the Gecode and ILOG libraries. Finally, some benchmarks show that the new search strategies actually improve the solving performance of $\mathcal{TOY}(\mathcal{FD})$.

Keywords: CFLP, FD Search Strategies, Solver Integration

1 Introduction

The use of *ad hoc* search strategies has been identified as a key point for solving Constraint Satisfaction Problems (CSP's) [Tsa93], allowing the user to interact with the solver in the search of solutions (exploiting its knowledge about the structure of the CSP and its solutions). Different paradigms provide different expressivity for specifying search strategies: Constraint Logic Programming CLP($\mathcal{FD}$) [JM94] and Constraint Functional Logic Programming CFLP($\mathcal{FD}$) [Han07] provide a declarative view of this specification, in contrast to the procedural one offered by Constraint Programming CP($\mathcal{FD}$) [MS98] systems (which make the programming of a strategy to depend on low-level details associated to the constraint solver, and even on the concrete machine the search is being performed). Also, due to their model reasoning capabilities, CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) treat search primitives as simple expressions, making possible to place a search primitive at any point of the program, combine several primitives to develop complex search heuristics, intermix search primitives with constraint posting, and use non-determinism to apply different search scenarios for solving a CSP.

* This work has been partially supported by the Spanish projects TIN2013-44742-C4-3-R, TIN2008-06622-C03-01, UCM-BSCH-GR58/08-910502, and S2009TIC-1465

The main contribution of this paper is to present a set of search primitives for CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems implemented in Prolog, and interfacing external CP($\mathcal{FD}$) solvers with a C++ API. The motivation of this approach is to take advantage of the high expressivity of CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) for specifying search strategies, and of the high efficiency of CP($\mathcal{FD}$) solvers. The paper focuses on the CFLP($\mathcal{FD}$) system $\mathcal{TOY}(\mathcal{FD})$ [FHSV07], more precisely in the system versions $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ [CS12] which interface the external CP($\mathcal{FD}$) solvers (with C++ API) of Gecode 3.7.3 [Gec] and IBM ILOG Solver 6.8 [ILO10], respectively. Regarding search, $\mathcal{TOY}(\mathcal{FD})$ offers two possibilities up to now. First, defining a new search from scratch at $\mathcal{TOY}(\mathcal{FD})$ level (using reflection functions to represent the search procedure). Second, use the search primitive `labeling`, which simply relies on predefined search strategies already existing in Gecode and ILOG, respectively. The use of external CP($\mathcal{FD}$) solvers (with C++ API) opens a third possibility, which is exploited in this work: Enhancing the search language of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ with new parametric search primitives, which are implemented in Gecode and ILOG by extending their underlying search libraries.

The paper is organized as follows: Section 2 presents a brief introduction to $\mathcal{TOY}(\mathcal{FD})$. Section 3 presents an abstract description of the new $\mathcal{TOY}(\mathcal{FD})$ search primitives. It points out some novel concepts not directly available in Gecode and ILOG Solver libraries. It also describes how to specify some search criteria at $\mathcal{TOY}(\mathcal{FD})$ level, and how easily these strategies can be combined to set different search scenarios. Section 4 describes the implementation of the primitives in $\mathcal{TOY}(\mathcal{FD})$, presenting an abstract view of the requirements, and how they are targeted to the Gecode and ILOG libraries. Section 5 presents some benchmarks, showing that the use of the search strategies improve the solving performance of both $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$. Section 6 presents some related work. Finally, Section 7 presents some conclusions and future work.

2 The $\mathcal{TOY}(\mathcal{FD})$ System

$\mathcal{TOY}(\mathcal{FD})$ (available at http://gpd.sip.ucm.es/ncasti/TOY(FD).zip) is implemented in SICStus Prolog [SIC]. It supports the solving of syntactic equalities and disequalities (via a host Herbrand solver: $\mathcal{H}$), and of $\mathcal{FD}$ constraints (via a connected CP($\mathcal{FD}$) solver). The $\mathcal{TOY}$ compiler uses SICStus Prolog as an object language [LLR93]. Its declarative semantics is based on a Conditional Term-Rewriting Logic: CRWL [GHLR99], and its operational semantics on a Constraint Lazy Narrowing Calculus: CLNC($\mathcal{FD}$) [LRV04].

A $\mathcal{TOY}(\mathcal{FD})$ *program* consists of a set of data constructors and a set of functions, that can be higher-order and non-deterministic (with possibly several reductions for given, even ground, arguments). The syntax is mostly borrowed from Haskell [PJ02], with the remarkable exception that program and type variables begin with upper-case letters whereas data constructors, types and functions begin with lower-case. The repertoire of $\mathcal{FD}$ constraints and operators is presented in Table 1, also including == and /=, as they are truly polymorphic (with the same operators for $\mathcal{H}$ and $\mathcal{FD}$).

The two syntactic domains *patterns* and *expressions* must be distinguished. Whereas an *expression* is susceptible of being reduced by the rules of the functions defined in the program, a

Type	Constraints and Operators
Relational Constraints	`==`, `/=`, `#>`, `#>=`, `#<`, `#<=`
Arithmetic Operators	`#+`, `#-`, `#*`, `#/`
Propositional Constraints	`post_implication`
Domain Constraint	`domain`, `domain_valArray`
Global Constraints	`all_different`, `count`, `sum`, `scalar_product`

Table 1: Repertoire of $\mathscr{FD}$ Constraints and Operators

pattern denotes a data value not subject of further evaluation (this includes variables, constants, data constructors and partial application of functions). A (user-)defined *function* is characterized by an optional principal type, which is checked/inferred by the system, and by a set of constrained rewriting rules $f\ t_1 \ldots t_n = e \ \Longleftarrow\ l_1 == r_1, \ldots, l_k == r_k$ where $t_1, \ldots, t_n$ form a tuple of linear patterns (i.e., with no repeated variables), and e, l_i, r_i are expressions. Rules have a conditional reading: $f\ t_1 \ldots t_n$ can be reduced to e if all the constraints $l_1 == r_1, \ldots, l_k == r_k$ are satisfied. For the case of non-deterministic functions, rules are applied following their textual order, and both failure and user request for a new solution trigger backtracking to the next unexplored rule.

A $\mathscr{TOY}(\mathscr{FD})$ *goal* consists of a set of constraints. Goal solving follows lazy narrowing: If a constraint is either an equality/disequality Herbrand constraint between patterns or a primitive finite domain constraint, then it is directly posted to its corresponding solver. Otherwise, the arguments of the constraint being expressions are lazily evaluated, applying matching function rules. This transforms the initial constraint into a primitive one, possibly producing more primitive or composed constraints to be processed. Once all the constraints of the goal have been processed, a $\mathscr{TOY}(\mathscr{FD})$ *solution* consists of the simplified $\mathscr{H}$ and $\mathscr{FD}$ constraint stores.

3 Search Primitives

This section presents eight new $\mathscr{TOY}(\mathscr{FD})$ primitives for specifying search strategies, allowing the user to interact with the solver in the search for solutions. Each primitive is presented from an abstract (solver independent) point of view, emphasizing some novel search concepts they provide. The specification of some search criteria at $\mathscr{TOY}(\mathscr{FD})$ level and the combination of primitives (to specify complex search strategies) are also presented.

3.1 Labeling Primitives

In this section, four search primitives are described: `lab`, `labB`, `labW` and `labO`.

Primitive lab

```
lab :: varOrd -> valOrd -> int -> [int] -> bool
```

This primitive collects (one by one) all possible combinations of values satisfying the set of constraints posted to the solver. It is parameterized by four basic components. The first and second ones represent the variable and value order criteria to be used in the search strategy, respectively.

```
myVarOrder:: [int] -> int
myVarOrder V = fst (foldl cmp (0,0)
                             (zip (take (length V) (from 0))
                                  (map (length . get_dom) V)))
%
myValOrder:: [[int]] -> int          | from:: int -> [int]
myValOrder D = head (last D)         | from N = [N | from (N+1)]
%
cmp:: (int,int) -> (int,int) -> (int,int)
cmp (I1,V1) (I2,V2) = if (V1 >= V2) then (I1,V1) else (I2,V2)
-----------------------------------------------------------------
TOY(FD)> domain [X,Y,Z] 0 4, Y /= 1, Y /= 3, Z /= 2,
         lab userVar userVal 2 [X,Y,Z], ... (rest of goal)
```

Figure 1: Variable and Value User-Defined Criteria

To express them we have defined in $\mathcal{TOY}$ the enumerated datatypes `varOrd` and `valOrd`, covering all the predefined criteria available in the Gecode documentation [STL13]. They also include a last case (`userVar` and `userVal`, respectively) in which the user implements its own variable/value selection criteria at $\mathcal{TOY}(\mathcal{FD})$ level. The third element `N` represents how many variables of the variable set are to be labeled. This represents a novel concept which is not available in the predefined search strategies of Gecode and ILOG Solver. The fourth argument represents the variable set `S`. Thus, the search heuristic uses `varOrd` to label just `N` variables of `S`.

Figure 1 presents a $\mathcal{TOY}(\mathcal{FD})$ program (top) and goal (bottom) showing how expressive, easy and flexible is to specify a search criteria in $\mathcal{TOY}(\mathcal{FD})$. In the example, the search strategy of the goal uses the `userVar` and `userVal` selection criteria (specified by the user in the functions `myVarOrder` and `myValOrder`, respectively). The `lab` search strategy is applicable to the constraint network posted by the $\mathcal{TOY}(\mathcal{FD})$ goal `domain [X,Y,Z] 0 4, Y /= 1, Y /= 3, Z /= 2`. Then, the computation continues by processing the "rest of goal" for each feasible solutions found by the `lab` strategy. It acts over the set of variables `[X,Y,Z]`, but it is only expected to label two of them.

The function `myVarOrder` selects first the variable with more intervals in its domain. It receives the list of variables involved in the search strategy, returning the index of the selected one. To this end, it uses the auxiliary functions `from` and `cmp`; the predefined functions `fst`, `foldl`, `zip`, `take`, `length`, `map`, `head`, `last` and `(.)` (all of them with an equivalent semantics to those of Haskell); and the reflection function `get_dom`, which accesses the internal state of the solver to obtain the domain of a variable (this domain is presented as a list of lists, where each sublist represents an interval of values).

The function `myValOrder` receives as its unique argument the domain of the variable, returning the lower bound of its upper interval. So, in conclusion, the first two solutions obtained by the `lab` strategy are: `{X in 0..4, Y -> 4, Z -> 3}` and `{X in 0..4, Y -> 4, Z -> 4}`.

Primitive labB

`labB :: varOrd -> valOrd -> int -> [int] -> bool`

This primitive uses the same four basic elements as `lab`. However, its behavior is different, as it follows the `varOrd` and `valOrd` criteria to explore just one branch of the search tree, with no backtracking allowed. The 4-Queens problem is used to explain this behavior.

Using `lab unassignedLeftVar smallestVal 0 [X1,X2,X3,X4]` (where `0` in the third argument stands for labeling all the variables) two solutions are obtained: `{X1 -> 1, X2 -> 3, X3 -> 2, X4 -> 4}` and `{X1 -> 2, X2 -> 4, X3 -> 1, X4 -> 3}`. However, if `labB unassignedLeftVar smallestVal 0 [X1,X2,X3,X4]` is used, then the strategy fails, getting no solutions. Figure 2 (4×4 square board and tree) shows the computation process. First, the selected criteria assigns `X1 -> 1` at root node (`1`), leading to node `2`. Propagation reduces the search space to `{X2 in 3..4, X3 in 2` $\vee$ `4, X4 in 2..3}`, pruning nodes `3` and `4`. Then, computation assigns `X2 -> 3` (leading to node `5`), and propagation leads to an empty domain for `X3`. So, the explored tree path leads to no solutions as well as, therefore, its computation. As it is seen, propagation during search modifies the intended branch to be explored (in the goal example, it explores the branch `1-2-5` instead of `1-2-3`).

Primitive labW

`labW :: varOrd -> bound -> int -> [int] -> bool`

This primitive performs an exhaustive breadth exploration of the search tree, storing the satisfiable leaf nodes achieved to further sort them by a specified criteria. A first example is considered to understand the behavior of `labW`. Figure 3 presents a $\mathcal{TOY}(\mathcal{FD})$ goal with four variables, where two implication constraints relate `X` and `Y` with `V1` and `V2`, respectively.

If `lab unassignedLeftVar smallestVal 2 [X,Y,V1,V2]` strategy had been used (instead of the `labW` one) to label the first two unbound vars of `[X,Y,V1,V2]`, then the search would have explored the search tree obtaining (one by one) the next four feasible solutions: `{X -> 0, Y -> 0}`, `{X -> 0, Y -> 1}`, `{X -> 1, Y -> 0}` and `{X -> 1, Y -> 1}`. Figure 4 represents the exploration for obtaining those solutions, where each black node represents a solution, and the triangle it has below represents the remaining size of the search space (product of cardinalities of `V1` and `V2`). As it is seen, whereas the first solution computed by `lab` leads to compute the "rest of goal" from a 12 candidates search space, the third solution leads to a 6 candidates one. The primitive `labW` explores exhaustively the search tree in breadth, storing in a data structure `DS` each feasible node leading to a solution. Once the tree has been completely explored, the solutions are obtained (one by one) by using a criteria to select and remove the *best* node from `DS`. In the example, the selected criteria `smallestSearchTree` selects first the node with smaller product of cardinalities of `V1` and `V2` (returning first the solution of the 6 candidates). The order in which the `labW` strategy of the goal delivers the solutions

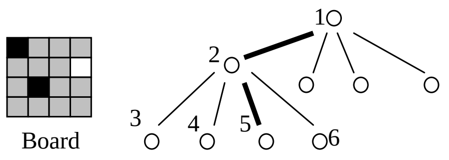

Figure 2: Applying `labB` to the Queens problem

```
TOY(FD)> domain [X,Y] 0 1, post_implication X (#=) 1 V1 (#>) 1,
         domain [V1,V2] 0 3, post_implication Y (#=) 0 V2 (#>) 0,
         labW unassignedLeftVar smallestSearchTree 2 [X,Y,V1,V2],
         ... (rest of goal)
```

Figure 3: `labW` Example

is presented in Figure 4.

Coming back to the definition of `labW`, the first parameter represents the variable selection criteria (no value selection is necessary, as the search would be exhaustive for all the values of the selected variables). The second parameter represents the *best* node selection criteria. To express it in $\mathcal{TOY}(\mathcal{FD})$, the enumerated datatype `ord` has been defined, ranging from the smallest/largest remaining search space of the product cardinalities of the labeling/solver-scope variables. Again, a last case (`userBound`) allows to specify the bound criteria at $\mathcal{TOY}(\mathcal{FD})$ level. The third parameter sets the breadth level of exhaustive exploration of the tree (represented as a horizontal black line in Figure 4). Finally, as usual, the last parameter is the set of variables to be labeled.

Figure 5 presents a $\mathcal{TOY}(\mathcal{FD})$ program (top) and goal (bottom) with a bound criteria specified in the user function `myBound`. The *best* node procedure selection traverses all the obtained nodes in `DS`, selecting first the one with minimal bound value. In this context, the user criteria specified in `myBound` assigns to each node (minus) the number of its singleton value search variables. Once again, the function `myBound` also relies on auxiliary, predefined and reflection functions. The first two obtained solutions are `{X -> 1, Y -> 1, A -> 0, B -> 0, C -> 0}` and `{X -> 2, Y -> 1, A in 0..1, B -> 0, C -> 0}`, respectively.

In summary, `labW` represents a novel concept which is not available in the predefined search strategies of Gecode and ILOG Solver. However, it must be used carefully, as exploring the tree very deeply can lead to a explosion of feasible nodes, producing memory problems for `DS` and becoming very inefficient (due to the time spent on exploring the tree and selecting the *best* node).

Primitive labO

`labO :: optType -> varOrd -> valOrd -> int -> [int] -> bool`

This primitive performs a standard optimization labeling. The first parameter `optType` contains the optimization type (minimization/maximization) and the variable to be optimized. The other four parameters are the same as in the `lab` primitive.

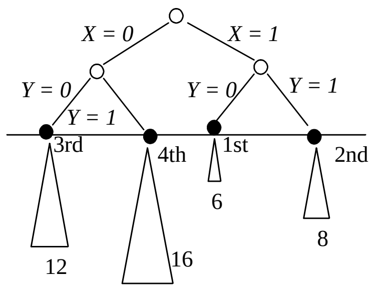

Figure 4: `labW` Search Tree Exploration

3.2 Fragmentize Primitives

```
frag :: domFrag -> varOrd -> intervalOrd -> int -> [int] -> bool
fragB:: domFrag -> varOrd -> intervalOrd -> int -> [int] -> bool
fragW:: domFrag -> varOrd -> bound -> int -> [int] -> bool
fragO:: domFrag -> optType->varOrd->intervalOrd->int->[int]-> bool
```

These four new primitives are mate to the labeling ones, but each variable is not labeled (bound) to a value, but *fragmented* (pruned) to a subset of the values of their domain. An introductory example is used to motivate the usefulness of these new primitives: A goal contains `V` variables and `C` constraints, with `V'` ≡ {`V1`, `V2`, `V3`} a subset of `V`. The constraint `domain V' 1 9` belongs to `C`. And no constraint of `C` relates the variables of `V'` by themselves, but some constraints relate `V'` with the rest of variables of `V`.

Figure 6 presents the search tree exploration achieved by `frag*` and `lab*` search primitives, respectively, where search nodes have been numbered. In the case of `frag*`, the three variables of `V'` have been fragmented into the intervals (`1`,...,`3`), (`4`,...,`6`) and (`7`,...,`9`), leading to exponentially less leaf nodes (27) than the `lab*` exploration (729). On the one hand, if it is known that there is only one solution to the problem, the probabilities of finding the right combination of `V'` values is thus bigger in `frag*` than in `lab*`. On the other hand, the remaining search space of the leaf nodes of `lab*` are expected to be exponentially smaller than the ones of `frag*`, due to the more propagation in `V'` (also expecting to lead to more pruning in the rest of variables `V`). Thus, the `frag*` search strategies can be seen as a more conservative technique, where there are less expectations of highly reducing the search space, as variables are not bound, but there is more probability of choosing a subset containing values leading to solutions (in what can be seen as a sort of generalization of *first-fail*). Coming back to the definition of each `frag*` primitive, two main differences arise w.r.t. its mate `lab*` primitive: First, it contains as an extra basic component (first argument) the datatype `domFrag`, which specifies the way the selected variable is fragmented. The user can choose between `partition n` and `intervals`. The former fragments the domain values of the variable into `n` subsets of the same cardinality. The latter looks for already existing intervals on the domain of the variables, splitting the domain on them. For example, in the goal `domain [X] 0 16, X /= 9, X /= 12` whereas applying `partition 3` to `X` fragments the domain in the subsets `S1` ≡ {0...4}, `S2` ≡ {5...8}∪{10} and `S3` ≡ {11}∪{13...16}, applying `intervals` fragments the domain in the subsets `S1'` ≡

```
isBound:: [[int]] -> bool
isBound [[A,A]] = true
isBound [[A,B]] = false <== B /= A
isBound [[A,B] | RL] = false <== length RL > 0
%
myBound:: [int] -> int
myBound V = - (length (filter isBound (map get_dom V)))
---------------------------------------------------------
TOY(FD)> domain [X,Y] 1 2, domain [A,B,C] 0 5,
         A #< X, B #< Y, C #< Y,
         labW unassignedLeftVar userBound 2 [X,Y,A,B,C]
```

Figure 5: Bound User-Defined Criteria

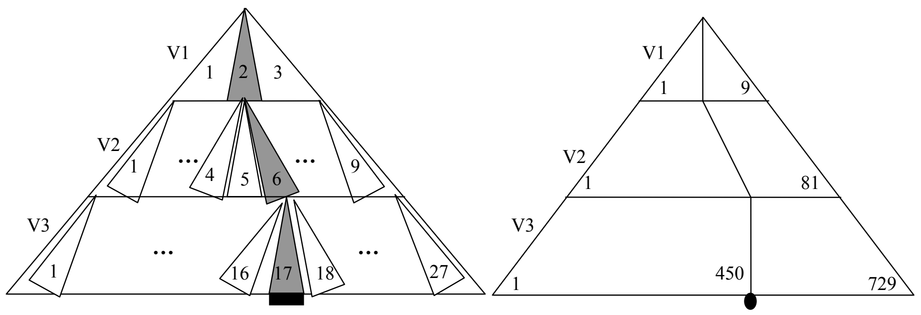

Figure 6: `frag` vs. `lab` Search Tree

$\{0...8\}$, `S2'` $\equiv \{10...11\}$ and `S3'` $\equiv \{13...16\}$. As a second difference, it contains an enumerated datatype `intervalOrd` (replacing the `lab⋆` argument `valOrd`), to specify the order in which the different intervals should be tried: First left, right, middle or random interval.

In summary, it is claimed that `frag⋆` primitives are an remarkable tool, to be taken into account in the context of search strategies as an alternative or a complement to the use of exhaustive labelings. Also, its use in $\mathcal{TOY}(\mathcal{FD})$ represents a novel concept which is not available in the predefined search strategies of Gecode and ILOG Solver.

3.3 Applying Different Search Scenarios

The use of $\mathcal{TOY}(\mathcal{FD})$ non-deterministic functions allows to sequentially apply different search strategies for solving a problem. For example, after posting `V` and `C` to the solver, the $\mathcal{TOY}(\mathcal{FD})$ program (top) and goal (bottom) presented in Figure 7 uses the non-deterministic function `f` to specify three different scenarios for the solving of the goal described in Section 3.2. Each scenario ends with an exhaustive labeling of the set of variables `V`. However, the search space s this exhaustive labeling has to explore can be highly reduced by the previous evaluation of `f`.

Scenario 1: The first rule of `f` performs the search heuristic h_1 over `V'` $\equiv$ {`V1`,`V2`,`V3`}. h_1 fragments the domain of `V1` into 4 subsets, selecting one randomly. If propagation succeeds, then h_1 bounds `V2` and `V3` to their smallest value. If propagation succeeds (with a remaining search space s_1), then h_1 succeeds, and the exhaustive labeling explores s_1. If propagation fails in one of those points, or the exhaustive labeling does not find any solution in s_1, then h_1 completely fails

```
f:: [int] -> bool
f [V1,V2,V3] = true <==
  fragB (partition 4) unassignedLeftVar random 0 [V1],
  labB unassignedLeftVar smallestVal 0 [V2,V3]
f [V1,V2,V3] = true <==
  fragW (partition 4) unassignedLeftVar smallestTree 0 [V1],
  labB unassignedLeftVar smallestTotalVars 0 [V2,V3]
f [V1,V2,V3] = true
--------------------------------------------
TOY(FD)> Post of (V,C), f V', lab userVar userVal 0 V
```

Figure 7: Applying Different Search Strategies

(as well as the first rule of `f`), as both the `labB` and `fragB` primitives just explore one branch.

Scenario 2: The second rule of `f` is tried, performing the heuristic h_2 over `V'`. Here a `fragW` primitive is first applied. So, if further either `labB` of h_2 or the exhaustive `lab` (acting over s_2) fails, backtracking is performed over `fragW`, providing the next *best* interval of `V1` (according to the smallest search tree criteria, as in Figure 4). If, after trying all the intervals a solution is not found, then h_2 completely fails (as well as the second rule of `f`).

Scenario 3: If both h_1 and h_2 fail, the third rule of `f` trivially succeeds, and the exhaustive labeling is performed over the original search space obtained after posting `V` and `C` to the solver.

4 Implementing the Search Primitives

The implementation of the eight new search primitives is based on the Gecode and ILOG Solver underlying search mechanisms. First, an abstract specification of the requirements the new $\mathcal{TOY}(\mathcal{FD})$ search strategies must fulfill is presented. Then, it is described how to adapt those requirements to Gecode and ILOG Solver.

4.1 Abstract Specification of the Search Strategy

A single entry point (C++ function) for the different primitives is specified. Its proposed algorithm is parameterizable by the primitive type and its basic components. It is described as follows:

1. The algorithm explores the tree by iteratively selecting a variable `var` and a value `v`, creating two options: (a) Post `var == v`. (b) Post `var /= v` to continue the exploration taking advantage of the previously explored branch, recursively selecting another value to perform again (a) and (b).

2. For `frag*` strategies it selects an interval `i` instead of a value, posting in (a) both `var #>= i.min` and `var #<= i.max`. However, the (b) branch can not take advantage by posting `var #< i.min` and `var #> i.max`, as the constraint store would become inconsistent. Thus, (b) just removes `i` from the set of intervals, and continue the search by selecting a new interval.

3. For `labB` and `fragB` strategies, only the (a) option is tried.

4. For `labO` and `fragO` strategies, branch and bound techniques are used to optimize the search.

5. Specific functions are devoted to variable and value/interval selection strategies, as well as to the bound associated to a particular solution found by `labW` and `fragW`. Those functions include the possibility of accessing Prolog, to follow the criteria the user has specified at $\mathcal{TOY}(\mathcal{FD})$ level (using $\mathcal{TOY}(\mathcal{FD})$ functions which are compiled to mate Prolog predicates).

6. The primitives `labW` and `fragW` perform the breadth exploration of the upper levels of the search tree, storing all the satisfiable leaf nodes to further provide them (one by one)

on demand. Thus, `ss` contains an entity performing the search and a vector `DS` (cf. Section 3.2) containing the solutions. The notion of solution is abstracted as the necessary information to perform the synchronization from `ss` to the main constraint solver. Also, a status indicates whether the exploration has finished or not.

7. The algorithm finishes (successfully) as it founds a solution, except for `labW` and `fragW` strategies, where it stores the solution node and triggers an explicit failure, continuing the breadth exploration of the tree.

8. A counter is used to control that only the specified amount of variables of the variable set is labeled/pruned.

Next two sections adapt this specification to Gecode and ILOG Solver, respectively. Table 2 summarizes the different notions provided by both libraries.

4.2 Gecode

Search strategies in Gecode are specified via `Branchers`, which are applied to the constraint solver (`Space`) to define the shape of the search tree to be explored. The `Space` is then passed to a `Search Engine`, whose execution method looks for a solution by performing a depth-first search exploration of the tree. This exploration is based on cloning `Spaces` (two `Spaces` are said to be equivalent if they contain equivalent stores) and hybrid recomputation techniques to optimize the backtracking. As `Spaces` constitute the nodes of the search tree, a solution found by the `Search Engine` is a new `Space`. The library allows to create a new class of `Brancher` by defining three class methods: `status`, which specifies if the current node is a solution, or their children must be generated to continue with their exploration; `choice`, which generates an object `o` containing the number of children the node has, as well as all the necessary information to perform their exploration; `commit`, which receives `o` and the concrete children identifier to perform its exploration (generating a new `Space` to be placed at that node).

Adaptation to the Specification. The search strategies are implemented via two layers. First, a new class of `Brancher MyGenerate`, which carries out the tree exploration by the combination of the `status`, `choice` and `commit` methods. As each node of the tree is a `Space`, the methods are applied to it. Second, a `Search Engine`, controlling the search by receiving the

Search Concept	Gecode	ILOG Solver
Search trigger	`Search Engine`	`IloGoal` stack
Tree node	`Space`	`IloGoal` attributes
Node exploration	`Brancher Commit`	`IloGoal` execution
Child generation	`Brancher Choice`	`IloGoal` constructor
Solution check	`Brancher Status`	Stack with no `IloAnd`
Solution abstraction	`Space`	Tree path (var,value) vector

Table 2: Different Search Concept Abstractions in Gecode and ILOG Solver

initial `Space` and making the necessary clones to traverse the tree. In this setting, the abstract description presented before is instantiated to Gecode as follows:

1. The `choice` method deals with the selection of the variable `var` and the value `v`, creating an object `o` with them as parameters, as well as the notion of having two children. The variable selection must rely on an external register `r`, being controlled by the `Search Engine` and thus independent on the concrete node (`Space`) the `choice` method is working with. The register is necessary to ensure that, whether a father generates its right hand child by posting `var /= v`, this child will reuse `r` to select again `var` (as a difference to the left hand child, which removes the `r` content to select a new variable).

2. For `frag*` strategies, instead of passing `val` to `o`, the `choice` method generates a vector with all the different intervals to be tried, and the size of this vector is passed as its number of children.

3. For `labB` and `fragB`, only one child is considered.

4. For `labO` and `fragO`, a specialized branch and bound `Search Engine` provided by Gecode is used.

6 The search entity is the `Search Engine` and the solution is a `Space`.

7 For `labW` and `fragW`, the `Search Engine` uses a loop, requesting solutions one by one until no more are found (the breadth exploration of the search tree has finished).

8 Only the left hand child of `lab*` strategies increments the counter value, and the `status` method checks the counter to stop the search at the precise moment.

4.3 ILOG Solver

Search strategies in ILOG Solver are performed via the execution of `IloGoals`. An `IloGoal` is a daemon characterized by its constructor and its execution method. The constructor creates the goal, initializing its attributes. The execution method triggers the algorithm to be processed by the constraint solver (`IloSolver`), and can include more calls to goal constructors, making the algorithm processed by `IloSolver` to be the consequence of executing several `IloGoals`. An `IloGoal` fails if `IloSolver` becomes inconsistent by running its execution method; otherwise the goal succeeds. The library allows to create a new class of `IloGoal` by defining its constructor and execution method. Four basic goal classes are provided for developing new goals with complex functionality. Goals `IlcGoalTrue` and `IlcGoalFalse` make the current goal succeed and fail, respectively. Goals `IlcAnd` and `IlcOr`, both taking two subgoals as arguments, make the current goal succeed depending on the behavior of its subgoals. While `IlcAnd` succeeds only if its two subgoals succeed, `IlcOr` creates a restorable choice point which executes its first subgoal, restores the solver state at the choice point on demand, and executes its second subgoal.

Adaptation to the Specification. The search strategies are implemented via the new `IloGoal` classes `MyGenerate` and `MyInstantiate`. Whereas the former deals with the selection of a variable, the latter deals with its binding/prunning to a value/interval. In this setting, the abstract description presented before is instantiated to ILOG Solver as follows:

1. The control of the tree exploration is carried out by `MyGenerate`, which selects a variable and uses the recursive call `IlcAnd(MyInstantiate, MyGenerate)` to bind it and further continue processing a new variable. In `MyInstantiate`, the alternatives (a) and (b) are implemented with `IlcOr(var == val, IlcAnd( var /= var, MyInstantiate))`.

2. It dynamically generates a vector with the available intervals on each different `MyGenerate` call.

3. Only the goal `var == val` is tried.

4. The branch and bound is explicitly implemented. Thus, before selecting each new variable, it is checked if the current optimization variable can improve the bound previously obtained; otherwise an `IloGoalFail` is used to trigger backtracking (as well as if, after labeling the required variables, the obtained solution does not bind the optimization variable).

6 The entity performing the search is an `IloSolver`. Also, a solution is represented by a vector of integers (representing the indexes of the labeled/pruned variables) and a vector of pairs, representing the assigned value or bounds of these variables. This explicit solution entity is built towards the recursive calls of `MyGenerate`, which adds on each call the index of the variable being labeled. Once found the solution, it stores it in `DS`.

7 After storing a solution in `labW` or `fragW`, an `IloGoalFalse` is used, triggering backtracking to continue the breadth exploration.

8 Each call to `MyGenerate` increments the counter value.

5 Performance

This section analyzes the new performance achieved by $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$. The benchmark includes four of the CP($\mathcal{FD}$) problems available at CSPLib [HMGW]: *Magic Sequence*, *N-Queens*, *Langford's number* and *Golomb Rulers*. The set of problems is claimed to be representative enough because: First, all are parametric, and thus they allow to test the performance of the $\mathcal{TOY}(\mathcal{FD})$ versions as the instances of each problem scale. And, second, they include the whole set of $\mathcal{FD}$ constraints of the $\mathcal{TOY}(\mathcal{FD})$ repertoire.

The structure of the solutions of each problem is discussed, pointing out how the new search strategies reduce the search exploration to find them. Thus, for each problem, two $\mathcal{TOY}(\mathcal{FD})$ models are created: *problem_bs.toy*, which applies a single `labeling` primitive as its search strategy; *problem_is.toy*, which applies some of the new proposed search primitives before applying the ending `labeling` (to still guarantee completeness of the search process).

Benchmarks are run in a machine with an Intel Dual Core 2.4Ghz processor and 4GB RAM memory. The OS used is Windows 7 SP1. The SICStus Prolog version used is 3.12.8. Microsoft Visual Studio 2008 tools are used for compiling and linking the $\mathcal{TOY}(\mathcal{FD}i)$ and

$\mathcal{TOY}(\mathcal{FD}g)$ C++ code. All the $\mathcal{TOY}(\mathcal{FD})$ models are available at: `http://gpd.sip.ucm.es/ncasti/models.zip`. For the sake of simplicity, from now on the different versions of the models will be simply referred to as *bs* and *is*.

5.1 Analyzing the Applied Search Strategies

Magic Sequence. The *bs* model uses a single `labeling [ff] L` as its search strategy. Analyzing the solutions of the problem it is observed that, if the parameter $n \geq 9$ then the sequence follows the pattern: $L \equiv [(n-4), 2, 1, 0,\ 0, \ldots, 1, 0, 0, 0]$. In this context, the new search strategy of the *is* model first applies `labB unassignedRightVar smallestVal 3 L, labB unassignedRightVar largestVal 1 L`, which matches the last four variable $1,0,0,0$ pattern. At that point, propagation leads to $L \equiv [(n-4), A, B, C, 0, \ldots, 1, 0, 0, 0]$ (with A in $1..3$, B in $0..1$ and C in $0..1$), highly reducing the search space the further `labeling` has to deal with.

Queens. The *bs* model uses a single `labeling [ff] L` as its search strategy. Analyzing the solutions of the problem, an intuitive way for reducing the initial search space of the problem consists of: First, splitting the n variables into k variable sets $(vs_1, vs_2, \ldots, vs_k)$ (where consecutive variables are placed in different variable sets). Second, splitting the initial domain $1 \ldots n$ into k different intervals $(1..(n/k), \ldots, (n/k)*(i-1)+1..(n/k)*i, \ldots, (n/k)*(k-1)+1..n)$. And, finally, assigning the variables of vs_i to the i^{th} interval.

In this context, the new search strategy of the *is* model first applies `split_into_3 L ([], [], []) == (K1,K2,K3), fragB (partition 3) unassignedLeftVar firstRight 0 K1, fragB (partition 3) unassignedLeftVar firstMiddle 0 K2, fragB (partition 3) unassignedLeftVar firstLeft 0 K3`. This splits the variables and their domains into three sets, highly reducing the search space the further `labeling` has to deal with.

Langford's Number. The *bs* model uses a single `labeling [ff] L` as its search strategy. Analyzing the solutions of the instances proposed, it is observed that they follow the pattern: $L \equiv [X1, X2, \ldots, A, B, C, D, E, F]$, with an inductive mapping between the set of variables {`A,B,E,F`} and the set of values {1,2,3,4}. In this context, the new search strategy of the *is* model first applies `fragB (partition ((round ((M*N)/4)) - 1)) unassigned-RightVar firstLeft 0 [A,B,E,F], labW unassigned- RightVar smallest-TotalDomain 0 [A,B,E,F]`.

The `fragB` fragments the domain of `[A,B,E,F]` in the `(M*N)/4` intervals of values `1..4`, `5..8`, ..., `M*N-3..M*N`. It selects the first interval starting from the left (i.e., the smallest one), and it precludes any further backtracking to explore the remaining intervals. Then, with the domain of `[A,B,E,F]` pruned to be in `1..4`, `labW` labels them, exploring all their feasible combinations before selecting the one leading to the smallest search space for `L`. Thus, it is clear that the use of the previous `fragB` is crucial for the success of the `labW` strategy. A deep breath exploration with `labW` implies a tradeoff between obtaining an ordered hierarchy of relevant intermediate tree-level nodes and the computational effort to obtain this hierarchy. With an initial domain of `1..M*N`, the feasible combinations of values for `[A,B,E,F]` is unaffordable in terms of time and memory. However, with a domain of `1..4` (and knowing that they are constrained with an $all\_different$) the amount of feasible combinations is reduced to, at most, 24 (which is clearly affordable).

Golomb Rulers. The *bs* model uses a single `labeling [toMinimize Mn] M` as its search strategy. Analyzing the solutions of the instances proposed, it is observed that the initial domain of their variables is huge, and that the value they take in the optimal solution is not far away from their initial lower bound. For example, in G-11 (an instance benchmark for 11 rulers, for which $M \equiv [0, A, B, ..., H, I, J]$), the initial domains of the last three variables are H in 36..1020, I in 45..1021 and J in 55..1023 (with known optimal solution 64, 70 and 72, respectively) and the initial domain of the first three variables is 0, A in 1..977 and B in 3..987 (with known optimal solution 0, 1 and 4, respectively).

In this context, an intuitive way of reducing the initial search space is by reducing as much as possible the upper bound of these variables. The new search strategy of the *is* model first applies `fragW (partition 3) unassignedRightVar smallestSearchTree 2 L`, `fragW (partition 12) unassignedLeftVar largestSearchTree 2 L`, fragmenting first the last two variables and then the first two. Note that, whereas the former selects as *best* intermediate node the one minimizing the remaining search space, the latter selects the one maximizing it (which intuitively makes sense, as the smaller interval is the one pruning the least the upper bound of the first two variables, thus pruning less the search space).

5.2 Running the Experiments

Table 3 compares the performance of mate *bs* and *is* instances. Columns 2 and 4 represent the CPU solving time in seconds of *bs* and *is* (respectively), both of them using incremental propagation mode. Columns 3 and 5 represent the speed-up of $\mathcal{TOY}(\mathcal{FD}g)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}i)$ for *bs* and *is*, respectively. Finally, column 6 focuses on each concrete $\mathcal{TOY}(\mathcal{FD})$ version, representing the speed-up of *is* w.r.t. *bs*. Some conclusions are obtained:

First, the use of the new search strategies is encouraging, as the performance of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ for solving *is* instances is better than the achieved for solving *bs* ones (excepting Q-90 and L-119, where $\mathcal{TOY}(\mathcal{FD}i)$ spends about 0.4 seconds more in solving *is*). In any case, the differences range from a 5% to nearly the 100%, so a more detailed analysis by problems and instances is required.

For Queens and Langford's *is* instances, the better performance achieved by $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ clearly scales as the sizes of the instances scale. More specifically, for Q-90 and L-119 (solved in tenths of seconds) $\mathcal{TOY}(\mathcal{FD}g)$ achieves an improvement of 22% and 5%, respectively. This improvement grows an order of magnitude for Q-105 and L-127 (with an improvement of 92%) and two orders of magnitude for Q-120 and L-131 (with an improvement of nearly the 100%). In $\mathcal{TOY}(\mathcal{FD}i)$, it is observed the same growing pattern, but it is less noticeable. For Q-90 and L-119 the *is* performance is even worse than the *bs* one. Then, for Q-105 and L-127 the *is* performance improves a 38% and a 63%, respectively, but still in the same order of magnitude as *bs*. Finally, for Q-120 and L-131 the *is* performance reaches the two orders of magnitude improvement w.r.t. *bs*, reaching nearly a 100%.

For Magic *is* instances, the better performance achieved by $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ remains stable as the size of the instances scale (with around a 33%-34% for $\mathcal{TOY}(\mathcal{FD}g)$ and a 23%-24% for $\mathcal{TOY}(\mathcal{FD}i)$). Last, for Golomb *is* instances the better performance decreases a 20% per instance (as they scale), with a 75%, 60% and 41% improvement of $\mathcal{TOY}(\mathcal{FD}g)$ for G-9, G-10 and G-11, respectively, and a 74%, 59% and 39% of $\mathcal{TOY}(\mathcal{FD}i)$.

Instance	bs	Sp-Up	is	Sp-Up	on/off
M-400 FDi	0.530	1.00	0.402	1.00	0.76
M-400 FDg	0.422	0.80	0.280	0.70	0.66
M-900 FDi	2.53	1.00	1.95	1.00	0.77
M-900 FDg	2.00	0.79	1.34	0.69	0.67
Q-90 FDi	0.110	1.00	0.514	1.00	4.67
Q-90 FDg	0.078	0.71	0.061	0.12	0.78
Q-105 FDi	1.25	1.00	0.78	1.00	0.62
Q-105 FDg	1.05	0.84	0.08	0.10	0.08
Q-120 FDi	154.00	1.00	1.11	1.00	0.01
Q-120 FDg	129.88	0.84	0.09	0.08	0.00
L-119 FDi	0.530	1.00	0.984	1.00	1.86
L-119 FDg	0.296	0.56	0.282	0.29	0.95
L-127 FDi	4.35	1.00	1.17	1.00	0.27
L-127 FDg	4.62	1.06	0.39	0.33	0.08
L-131 FDi	87.00	1.00	1.19	1.00	0.01
L-131 FDg	98.53	1.13	0.33	0.28	0.00
G-9 FDi	0.421	1.00	0.109	1.00	0.26
G-9 FDg	0.250	0.59	0.062	0.57	0.25
G-10 FDi	3.56	1.00	1.47	1.00	0.41
G-10 FDg	2.11	0.59	0.84	0.57	0.40
G-11 FDi	72.65	1.00	43.98	1.00	0.61
G-11 FDg	42.01	0.58	24.85	0.57	0.59

Table 3: Performance of $\mathcal{TOY}(\mathcal{FD})$ using the Search Strategies

Second, it is clearly observed that the improvement achieved by $\mathcal{TOY}(\mathcal{FD}g)$ for *is* instances is bigger than the one achieved by $\mathcal{TOY}(\mathcal{FD}i)$, revealing that the approach Gecode offers to extend the library with new search strategies is more efficient than the one of ILOG Solver. That is, for any *is* instance, the speed-up of $\mathcal{TOY}(\mathcal{FD}g)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}i)$ is bigger than the achieved for its mate *bs* instance.

In this context, two different behaviors are observed. First, for Queens and Langford's *is* instances the speed-up improvement achieved w.r.t. *bs* instances increases as the instances scale: A 59%, 74% and 76% for Q-90, Q-105 and Q-120, respectively. A 27%, 73% and 85% for L-119, L-127 and L-131, respectively. Second, for Magic and Golomb *is* instances the speed-up improvement achieved w.r.t. *bs* instances remains stable as the instances scale: A 10% for M-400 and M-900. A 2%, 2% and 1% for G-9, G-10 and G-11, respectively.

6 Related Work

The approach of taking advantage of both the high expressivity of $\mathcal{TOY}(\mathcal{FD})$ and of the high efficiency of Gecode and ILOG Solver can be related to the one followed in *Search Combinators*

[STW+13]. It provides a lightweight and solver-independent method bridging the gap between a conceptually simple search language (high level, functional and naturally compositional) and an efficient implementation (low-level, imperative and highly non-modular). $\mathcal{TOY}(\mathcal{FD})$ is more rigid than [STW+13], but some of the features provided by the search combinators can be matched with the new set of primitives presented in Section 3: The basic primitive heuristics `base_search` and `prune` can be obtained with the primitive `lab`, controlling the exact number of variables to be labeled (which allows $\mathcal{TOY}(\mathcal{FD})$ to support composite search strategies). Regarding the set of combinators proposed, $\mathcal{TOY}(\mathcal{FD})$ matches {`let`, `assign`, `post`} via intermixing search procedures with constraint posting, and {`and`, `or`} via the composed search strategies presented in Section 3.3. Finally, the $\mathcal{TOY}(\mathcal{FD})$ primitives are also extensible, as users can program their own criteria at $\mathcal{TOY}(\mathcal{FD})$ level with no extra effort at the SICStus and C++ core implementation of the system.

Similar approaches to search combinators are proposed for Constraint Funcional Programming (CFP($\mathcal{FD}$)), with Monadic Constraint Programming [SSW09], and for CLP($\mathcal{FD}$), with the library Tor [STD12] (available in SWI-Prolog). They decouple the definition of the search tree and the search method. In $\mathcal{TOY}(\mathcal{FD})$, it is not possible to specify the way to explore the search tree (as, for example, by limited discrepancy search). However, the primitives `labW` and `fragW` perform a breadth search exploration. Also, whereas these primitives include a depth bound, it can be implicitly imposed as well for the rest of primitives (by using the parameter setting the amount of variables to be labeled). Moreover, at least in $\mathcal{TOY}(\mathcal{FD}g)$ it would not be difficult to support new ways of exploring the search tree, as the Gecode library provide the mechanisms to implement them.

7 Conclusions and Future Work

This paper has presented eight new $\mathcal{TOY}(\mathcal{FD})$ search primitives, describing their behavior from a solver independent point of view, and using examples to show their application. It has emphasized the novel concepts those primitives include, as performing an exhaustive breadth exploration of the search tree further sorting the satisfiable solutions by a specified criteria, fragmenting the domains of the variables by pruning each one to a subset of its domain values instead of binding it to a single value, and applying the labeling or fragment strategy only to a subset of the variables involved. It has also pointed out how expressive, easy and flexible it is to specify some search criteria at $\mathcal{TOY}(\mathcal{FD})$ level, and also to apply different search strategies (setting different search scenarios) to the solving of a CP($\mathcal{FD}$) problem.

A new version of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ including these search primitives has been presented. It has been described their implementation in Gecode and ILOG Solver, by extending their libraries relying on their underlying search mechanisms. It has been observed that these search mechanisms are quite different in Gecode (`Search Engine`, `Brancher` methods, hybrid recomputation) and ILOG Solver (`IloGoal`, goal constructor, goal stack). Thus, an abstract view of the requirements needed to integrate the search strategies in $\mathcal{TOY}(\mathcal{FD})$ has been first presented (with the scheme further instantiated to Gecode and ILOG Solver).

Finally, standard benchmarks have been used to point out how the use of the proposed search strategies allow to reduce the search exploration to find them. Mate $\mathcal{TOY}(\mathcal{FD})$ models, either with a classical `labeling` and with an improved *ad hoc* search strategy have been developed. It has been proven that the use of the new search strategies improve the performance of both $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$, but the improvement achieved (ranging in 5%-100%) is dependent on the concrete problem and instance solved: Whereas for Queens and Langford's instances the better performance achieved clearly scales as the sizes of the instances scale, for Magic ones it remains stable, and for Golomb ones it decreases. Moreover, the speed-up of $\mathcal{TOY}(\mathcal{FD}g)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}i)$ is bigger for the new improved $\mathcal{TOY}(\mathcal{FD})$ models, revealing that the approach Gecode offers to extend the library with new search strategies is more efficient than the one of ILOG Solver.

As future work, we will analyze the applicability of the search strategies presented in this paper to other CP($\mathcal{FD}$) paradigms. In particular, we will implement the search primitives in the CFP($\mathcal{FD}$) system Monadic Constraint Programming, the CLP($\mathcal{FD}$) system Tor and the C++ CP($\mathcal{FD}$) system Gecode (using the search combinators [STW+13] to implement the strategies). We will discuss if there are aspects of $\mathcal{TOY}(\mathcal{FD})$ that are not easily implemented in the other systems, such as the use of non-deterministic functions (to apply different search scenarios to tackle a problem), and the specification of some search criteria in the proper native language (and the impact it may have in the system architecture). We will also reuse the benchmark used in this paper to compare the solving efficiency achieved by each system when applying the *ad hoc* strategies, analyzing any possible overhead arisen due to their use.

Besides that, focusing again in $\mathcal{TOY}(\mathcal{FD})$, scripting techniques can be applied, to solve the benchmarks under multiple and very precisely controlled scenarios. In them, an exhaustive combination of applying one or different search strategies (as well as the variable subset used on each of them) will be studied. The results will be analyzed, in order to find out which strategies had lead to a solution or, at least, to a minimum search space containing a solution. Moreover, this analysis will help to find out new patterns about the relation between the structure of a concrete problem and the concrete search strategy (or combination of search strategies) to be applied to successfully solve it.

Bibliography

[CS12] I. Castiñeiras, F. Sáenz-Pérez. Improving the Performance of FD Constraint Solving in a CFLP System. In *FLOPS'12, 88–103*. LNCS 7294. Springer, 2012.

[FHSV07] A. J. Fernández, T. Hortalá-González, F. Sáenz-Pérez, R. del Vado-Vírseda. Constraint Functional Logic Programming over Finite Domains. *TPLP* 7(5):537 – 582, 2007.

[Gec] Gecode: Generic Constraint Development Environment. Version 3.7.3. http://www.gecode.org/.

[GHLR99] J. González-Moreno, M. Hortalá-González, F. López-Fraguas, M. Rodríguez-Artalejo. An Approach to Declarative Programming based on a Rewriting Logic. *Journal of Logic Programming* 40:47–87, 1999.

[Han07] M. Hanus. Multi-Paradigm Declarative Languages. In *ICLP'07, 45–75*. LNCS 4670. Springer, 2007.

[HMGW] B. Hnich, I. Miguel, I. P. Gent, T. Walsh. CSPLib: A Problem Library for Constraints. http://www.csplib.org/.

[ILO10] IBM ILOG CP 1.6. 2010. http://www-947.ibm.com/support/entry/portal/Overview/Software/WebSphere/IBM_ILOG_CP.

[JM94] J. Jaffar, M. Maher. Constraint Logic Programming: A Survey. In *The Journal of Logic Programming, 19–20. 503–581*. Elsevier, 1994.

[LLR93] R. Loogen, F. López-Fraguas, M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *PLILP'93, 184–200*. LNCS 714, 1993.

[LRV04] F. López-Fraguas, M. Rodríguez-Artalejo, R. Vado-Vírseda. A Lazy Narrowing Calculus for Declarative Constraint Programming. In *PPDP'04, 43–54*. ACM, 2004.

[MS98] K. Marriot, P. J. Stuckey. *Programming with Constraints*. MIT Press, 1998.

[PJ02] S. Peyton-Jones. Haskell 98 Language and Libraries: the Revised Report. Technical report, 2002. http://www.haskell.org/onlinereport/.

[SIC] SICStus Prolog. http://www.sics.se/.

[SSW09] T. Schrijvers, P. Stuckey, P. Wadler. Monadic Constraint Programming. *Journal of Functional Programming* 19(6):663–697, 2009.

[STD12] T. Schrijvers, M. Triska, B. Demoen. Tor: Extensible Search with Hookable Disjunction. In *PPDP'12, 103–114*. ACM, 2012.

[STL13] C. Schulte, G. Tack, M. Z. Lagerkvist. Modeling and Programming with Gecode. 2013. http://www.gecode.org/doc-latest/MPG.pdf.

[STW$^+$13] T. Schrijvers, G. Tack, P. Wuille, H. Samulowitz, P. Stuckey. Search combinators. *Constraints* 18(2):269–305, 2013.

[Tsa93] E. Tsang. *Foundations of constraint satisfaction*. Academic Press, 1993.