A reactive and hybrid constraint solver

Eric Monfroy\textsuperscript{a,b}, Carlos Castro\textsuperscript{c}, Broderick Crawford\textsuperscript{d}, Ricardo Soto\textsuperscript{d,e}, Fernando Paredes\textsuperscript{f} & Christian Figueroa\textsuperscript{g}

\textsuperscript{a} Departamento de Informática, Universidad Técnica Federico Santa María, Av. España 1680, Valparaíso, Av. Chile
\textsuperscript{b} Département d’Informatique, Université de Nantes, France
\textsuperscript{c} Departamento de Informática, UTFSM, Valparaíso, Chile
\textsuperscript{d} Escuela de Ingeniería Informática, Pontificia Universidad Católica de Valparaíso, Chile
\textsuperscript{e} Universidad Autónoma de Chile, Chile
\textsuperscript{f} Escuela de Ingeniería Industrial, Universidad Diego Portales, Santiago, Chile
\textsuperscript{g} Zeke, Viña del Mar, Chile

Published online: 02 Feb 2012.


To link to this article: http://dx.doi.org/10.1080/0952813X.2012.656328

PLEASE SCROLL DOWN FOR ARTICLE

Taylor & Francis makes every effort to ensure the accuracy of all the information (the “Content”) contained in the publications on our platform. However, Taylor & Francis, our agents, and our licensors make no representations or warranties whatsoever as to the accuracy, completeness, or suitability for any purpose of the Content. Any opinions and views expressed in this publication are the opinions and views of the authors, and are not the views of or endorsed by Taylor & Francis. The accuracy of the Content should not be relied upon and should be independently verified with primary sources of information. Taylor and Francis shall not be liable for any losses, actions, claims, proceedings, demands, costs, expenses, damages, and other liabilities whatsoever or howsoever caused arising directly or indirectly in connection with, in relation to or arising out of the use of the Content.
A reactive and hybrid constraint solver

Eric Monfroy\textsuperscript{ab*}, Carlos Castro\textsuperscript{c}, Broderick Crawford\textsuperscript{d}, Ricardo Soto\textsuperscript{de}, Fernando Paredes\textsuperscript{f} and Christian Figueroa\textsuperscript{g}

\textsuperscript{a}Departamento de Informática, Universidad Técnica Federico Santa María, Av. España 1680, Valparaíso, Av. Chile; \textsuperscript{b}Département d’Informatique, Université de Nantes, France; \textsuperscript{c}Departamento de Informática, UTFSM, Valparaíso, Chile; \textsuperscript{d}Escuela de Ingeniería Informática, Pontificia Universidad Católica de Valparaíso, Chile; \textsuperscript{e}Universidad Autónoma de Chile, Chile; \textsuperscript{f}Escuela de Ingeniería Industrial, Universidad Diego Portales, Santiago, Chile; \textsuperscript{g}Zeke, Viña del Mar, Chile

(Received 21 April 2011; final version received 20 November 2011)

In Castro et al. [Castro, C., Monfroy, E., Figueroa, C., and Meneses, R. (2005), ‘An Approach for Dynamic Split Strategies in Constraint Solving’, in Proceedings of MICAI 2005 (Vol. 3789), LNAI, Berlin: Springer, pp. 162–174] a framework for adaptive enumeration strategies and meta-backtracks for a propagation-based constraint solver has been studied. Here, we extend this framework in order to trigger some functions of a solver, or of a hybrid solver to respond to some observations of the solving process. We can also simply design adaptive hybridisation strategies by just changing some rules of the update component of our framework. We experiment with this framework on a hybrid Branch and Bound \textsuperscript{+} propagation solver in which propagation can be triggered w.r.t. some observations of the solving process. The results show that some phases of propagation are not only beneficial to the Branch and Bound algorithm, but also that propagation is too costly to be executed at each node of the search tree. The hybridisation strategies are thus crucial in order to decide when to perform the or not propagation.

Keywords: constraint solving; hybrid solver; reactive search; autonomous search

1. Introduction

Over recent decades, impressive improvements have been achieved in solving complex combinatorial optimisation problems, issued from real world applications, and which involve more and more data and constraints. In order to tackle large-scale instances and intricate problem structures, sophisticated solving techniques have been developed, combined and hybridised to provide efficient solvers.

Combinatorial problems are often modelled as constraint satisfaction problems or constraint optimisation problems, which consist of a set of variables, a set of possible values for these variables and a set of constraints to be satisfied. However, solvers or hybridisation of solvers become more and more complex: the user must select or design various solving and hybridisation strategies and tune numerous parameters, either related...
to the solvers or to the hybridisation strategy. Moreover, it is well-known that an *a priori* decision concerning strategies and parameters is very difficult since the effects of strategies and parameters are rather unpredictable and may change during resolution.

The selection and the correct setting of the most suitable algorithm for solving a given problem has already been investigated many years ago by Rice (1975). The proposed abstract model suggested extracting features in order to characterise the problem, searching for a suitable algorithm in the space of available algorithms and then evaluating its performances with respect to a set of measures.

These considerations are still valid and can indeed be considered at least from two complementary points of view:

- selecting solving techniques or algorithms from a set of possible available techniques
- tuning an algorithm with respect to a given instance of a problem.

Some approaches have been developed to answer the algorithm selection problem in various fields as described in the recent survey of Smit-Miles et al. (2008).

Moreover, settings should be changed to adapt themselves and react to the process during solving.

An Autonomous Search (Hamadi, Monfroy, and Saubion 2012) system should provide the ability to advantageously modify its internal components when exposed to changing external forces and opportunities. It corresponds to a particular case of adaptive systems with the objective of improving its problem-solving performance by adapting its search strategy to the problem at hand. The literature about autonomous search is vast and many works are going on, e.g. Smit-Miles et al. (2008) for a generic approach, and some more ‘specialised’ works such as Lobo, Lima, and Michalewicz (2007) and Smit and Eiben (2009) for evolutionary algorithms, Battiti and Brunato (2010) and Battiti, Brunato, and Mascia (2008) for reactive search or O’Mahony et al. (2008) for case-based reasoning for constraint programming.

In Hamadi, Monfroy, and Saubion (2010), a general definition and a taxonomy of autonomous search processes with respect to their computation characteristics (solver that are able to modify or adjust their strategies and parameters either *a priori* or adaptively, self-adaptation or supervised adaptation, etc.) is given.

In Castro, Monfroy, and Figueroa (2005), a framework for dynamic adaptation of enumeration strategies of a propagation-based solver (see, e.g. Apt 2003) was proposed. This framework was then extended to handle some functions, such as a meta-backtrack function which enables the solving process to jump back several nodes in the search tree (Monfroy, Castro, and Crawford 2006).

Here, we extend this framework in order to manage dynamically some strategies and components of a solver or hybrid solver. Our framework can hence be used to manage reactively some solvers inside a hybridisation of solving processes. This is achieved by some rules that trigger some solvers or functions of solvers reacting to some observations of the search.

We instantiate and experiment on this framework by managing the hybridisation strategy of a (rather common) hybrid solver based on a Branch and Bound (B&B) algorithm combined with constraint propagation to reduce the search space. To this end, the framework we propose observes the resolution process, analyses the observations and makes some decisions to possibly dynamically adapt strategies and trigger some functions
of the solver. With respect to the classification of Hamadi et al. (2010), we propose an autonomous and supervised adaptive solving framework.

The results show that:

- it is rather easy to change the hybridisation strategy by just changing some rules in our framework;
- reactivity to some observations of the solving process enables one to design strategies that take into account the behaviour of the search;
- propagation is beneficial to the B&B algorithm, but also that it is too costly to be executed at each node of the search tree. The hybridisation strategies and reactivity are thus crucial in order to ‘tune’ when to perform the propagation.

This article is organised as follows. In Section 2 we give an overview of our framework for reactive handling of enumeration strategies and solver functions. Section 3 describes the hybrid solver based on B&B and constraint propagation. Several strategies are described with some few rules, and experiments of each strategy are shown and discussed. Finally, Section 4 presents a general discussion of the framework for hybridisation, a conclusion and some future works.

2. The supervised adaptive framework

Here, we present the main principles of our framework.

2.1. Overview of the framework

Our framework for adaptive and autonomous solvers (Figure 1) is based on four components that exchange information: the first component runs a solver or a solver cooperation/hybridisation based on some tunable modules/functions and on some solving strategies; the second one observes resolution and takes snapshots, i.e. some kind of quantitative summaries of the observations; the third one analyses snapshots and draws some indicators about strategy and function quality and the fourth one makes decisions to update strategy priorities and trigger functions.

This approach is based on two key features: rules to change strategy when a strategy behaves badly and when we guess another one could work better continuing deeper in the search tree and rules to trigger components or functions of the solver. In this article, we focus on triggering some functions of a hybrid solver with respect to the behaviour of the solving process.

Figure 1. The adaptive strategy framework.
2.2. The SOLVE component

The SOLVE component is a solver, or a hybrid solver which aims at solving constraint problems. The special feature of this (hybrid-)solver is that it has at disposal several candidate strategies and several functions that can be changed or triggered using some rules of the UPDATE component.

In Castro et al. (2005), we described a generic solving algorithm which alternates constraint propagation with enumeration phases: ‘bad’ strategies of enumeration were replaced by more promising ones, and some ‘meta-backtracks’ enabled to jump back of several enumerations and propagation phases when the solving state was becoming very bad. The jump-back mechanism is detailed in Monfroy et al. (2006).

We focus here on a hybrid solver, a part of which will be triggered by our rules with respect to some observations of the solving process. The algorithm is a B&B algorithm: while there remain sub-problems, choose one (depth-first selection in our case) and treat it. This algorithm (Algorithm 1) is hybrid (compared to a usual B & B algorithm), since each sub-problem \( P \) may be reduced by a propagation (e.g. Apt 2003) phase to reduce the search space by eliminating values of variables that cannot participate in a solution.

Algorithm 1: Sketch of the hybrid algorithm

```plaintext
1: Procedure HYBRID_B&B
2: Optimum ← −∞
3: Nodes ← (\( P_0 \), −∞)
4: while Nodes ≠ \( \emptyset \) do
5: \( P \leftarrow \) getSubproblem(Nodes)
6: PropagationPhase(\( P \)) (conditional)
7: RelaxationPhase(\( P \))
8: checkSolution(\( P \))
9: branching(\( P \), (\( P_1 \), ..., \( P_k \)))
10: for (1 ≤ i ≤ k) do
11: Nodes ← Nodes \( ∪ \) (\( P_i \), bound(\( P \)))
12: end for
13: OptimumSolution ← Solution
14: OptimumValue ← Optimum
15: end while
16: end Procedure
```

This technique is rather common (see, e.g. Rodosek, Wallace, and Hajian 1999), thus, we just give an overview of the method. We consider two models of the problem to be solved. The first one, \( M_{\text{CSP}} \) is based on constraint satisfaction problem (CSP) and uses finite domain variables (i.e. \( X_{\text{CSP}} \in \{v_1, ..., v_n\} \)), and the second one, \( M_{\text{IP}} \), is a linear programming model (i.e. based on continuous variables \( v_1 < X_{\text{IP}} < v_n \)). The two models are equivalent and represent the same information. At each node of the search tree, the problem is modelled by both techniques. Information discovered in a model is communicated to the other one (e.g. if \( X_{\text{CSP}} \) is reduced to the domain \( \{v_1, ..., v_n\} \), then the corresponding variable \( X_{\text{IP}} \) can be constrained by \( v_1 < X_{\text{IP}} < v_n \), and so on for objective functions and in the opposite way).
The (conditional) propagation phase is applied to the \( M_{\text{CSP}} \) model to try to reduce the search space by removing values of variables that cannot satisfy some constraints. The obtained information (reduction of domains) is communicated to the \( M_{\text{IP}} \) model; the relaxation phase is applied to the relaxation of the \( M_{\text{IP}} \) model (i.e. \( M_{\text{IP}} \) without integrity constraints). In case there is no propagation phase, the algorithm acts as a usual B & B algorithm: the relaxation is solved to determine the unsatisfiability of the problem or an integer solution.

Then, when checking solution, the algorithm evaluates whether the solution is feasible or not. If the solution is integer with a better value than the actual one, the optimum value and the optimal actual solution are updated. If the solution is integer, but not better, it is discarded. If it is not-integer, the branching is done on non-integer variables, generating the respective subproblems w.r.t. the enumeration strategy (in this case, after some preliminary tests, we fixed the lexicographic order to select the variable, and the largest value of the variable).

In the following, we use our framework to (de)activate the propagation phase: by just changing the updating rules we obtain different types of hybridisation.

### 2.3. The OBSERVATION component

The OBSERVATION component aims at observing and recording information of the current search tree, i.e. it spies the resolution process of the SOLVE component. These observations (called snapshots) can be seen as an abstraction of the resolution state at a time \( t \). Taking a snapshot consists in extracting (since search trees are too large) and recording some information from a resolution state. Thus, two main issues are important: when to take snapshots and which information to record.

Snapshots can be taken regularly (i.e. at every \( n \) ms, or every \( m \) loops of the SOLVE algorithm) or when some events happen (i.e. a variable was fixed, the search space was reduced of \( x \% \)).

The recorded information aims at reflecting resolution: it will be analysed, and used to update priorities of strategies or to trigger some functions of the hybrid solver. Snapshots can mainly contain three types of information: characteristics of the problem (e.g. hard variables of the problem, linear constraints, or occurrences of variables), measures of the search tree (e.g. current depth, maximal depth, fixed variables or size of the current search space) and properties of the computation (e.g. CPU time of a propagation phase or which operators were used).

In the experimentations we describe in this article, snapshots are taken at each node of the search tree and they contain the following data:

- **Characteristic of the problem:**
  - \( V_{\text{total}} \): total number of variables
  - \( R_{\text{eq}} \): number of constraints of type \( = \)
  - \( R_{\text{leq}} \): number of constraints of type \( \leq \)
  - \( R_{\text{geq}} \): number of constraints of type \( \geq \)

- **Measures of the search tree**
  - \( N_{\text{expl}} \): number of explored nodes
  - \( N_{\text{failed}} \): failed nodes (not leading to a solution)
The ANALYSE component analyses the snapshots taken by the OBSERVATION: it evaluates the strategies and functions, and provides indicators to the UPDATE component. Indicators can be Boolean or numeric values ($\delta b$ and $\delta n$, respectively), they can be extracted, computed, or deduced from one or several snapshots.

Numeric indicators ($\delta n$) are results of quantitative computations of measures recorded in snapshots. Simple indicators are the depth of the search ($\delta n_{\text{depth}}$), the number of fixed variables ($\delta n_{\text{fix}}$), or fixed by enumeration ($\delta n_{\text{fixen}}$), or the average size of domains ($\delta n_{\text{davg}}$). More complex indicators can be the difference of depth between 2 snapshots to give information on the evolution of the search tree. If this difference is large, a good progress was done; if it is small, the search can be stuck at some level. The difference between the depth ($\delta n_{\text{depth}}$) of the search and the variables fixed by enumeration ($\delta n_{\text{fixen}}$) gives an indicator ($\delta n_{\text{gap}}$) of how many unsuccessful enumerations were performed on the last variable (Castro et al. 2005). Indicators can also be related to efficiency in terms of time, e.g. the CPU time consumed by one of the solver functions between two snapshots. As an example, in Section 3.6, we will present a strategy based on the ratio of CPU time used by relaxation and propagation.

Boolean indicators ($\delta b$) reflect properties. Simple ones can be related to problems (e.g. there is a univariate constraint or a hard variable was fixed). More complex properties can be related to a quantitative analysis of the snapshots. For example, consider a solver based on constraint propagation and enumeration. Consider $n$ consecutive snapshots such that the number of instantiated variables oscillate with a small amplitude. We can deduce that the SOLVE component alternates enumerations and backtracks on several variables, without succeeding in having a strong orientation (e.g. going deeper in the search tree or performing a significant backtracking phase): this is a thrashing-type behaviour. Such an indicator was used in Castro et al. (2005).

The indicators that we used for our experimentation are described later on, for each hybridisation (Section 3).

The UPDATE component makes decisions using the indicators: it makes interpretations of the indicators, and then updates the strategies priorities and/or triggers some functions of the SOLVE component. The knowledge of the UPDATE component is contained in a
set of rules. The head of such a rule is a conjunction of conditions on the indicators (disjunctions can be handled by several rules). There are two types of rules: for priority update rules (⇒ rules), the body is a conjunction of updates of strategies priorities:

$$\bigwedge_{i=1}^{l} \left( \sum_{j \in J_i} \omega_j \times \delta n_j \right) \text{ op } c_j \land \bigwedge_{i=1}^{k} \delta b_i \Rightarrow \bigwedge_{i=1}^{l} p_i = p_i + f_i(\delta n_1, \ldots, \delta n_l)$$

where:

- the $\omega_j$ are the weights of each numeric indicator $\delta n_j$ in the condition, the $c_j$ are constants, the $J_i$ are subsets of all the indicators and $\text{op} \in \{\leq, \geq, =\}$;
- the $\delta b_i$ are some Boolean indicators;
- the $f_i$ are functions over the indicators that return real numbers to increase or decrease the priority $p_i$ of the strategy $i$;
- and the $\bigwedge_{i=1}^{l}$ in the body of the rule is an abuse of language which means that the $l$ priorities can be updated.

For function rules (→ rules) the body requests the application of a function of a solver, possibly with some parameters:

$$\bigwedge_{i=1}^{l} \left( \sum_{j \in J_i} \omega_j \times \delta n_j \right) \text{ op } c_j \land \bigwedge_{i=1}^{k} \delta b_i \rightarrow \text{ function(…)}$$

When the head of a rule is fulfilled (i.e. conditions are verified), its body is executed: for ⇒ rules, the priorities of the strategies (e.g. enumeration strategies) are updated in the SOLVE component. Whereas for → rules, functions are triggered in the SOLVE component.

The rules we used for our experimentations are described in the next section for each hybridisation. Some rules for updating strategies and triggering meta backtracks and restarts are shown in Castro et al. (2005) and Monfroy et al. (2006).

3. Experimentation: an adaptive hybrid B&B solver

We now describe experimentation with a hybrid B&B + propagation solver, as described in Figure 1. The snapshots, indicators and rules enable us to simply change the strategy of hybridisation. In this practical case, since we do not use priority of strategies, we only use rules of the second type that will activate (propagation()) or deactivate (nopropagation()) the propagation phase.

$$\bigwedge_{i=1}^{l} \left( \sum_{j \in J_i} \omega_j \times \delta n_j \right) \text{ op } c_j \land \bigwedge_{i=1}^{k} \delta b_i \rightarrow \text{(no) propagation()}$$

Thus, by just changing the updating rule, we obtain different types of hybridisation.

Our goal is not to design the ‘best’ solver, but to show how simply the hybridisation strategy can be changed with our framework, and to observe the role of propagation on various problems and thus to give some hints on how to manage it. We now describe the conditions of experimentation.
The solvers  The solving process was written in Gecode (Schulte, Tack, and Lagerkvist 2006); the propagation is achieved by the propagators of Gecode while the components of the B&B algorithm (e.g. relaxation) are achieved by lp_solve (lp_solve (Berkelaar) is a linear (integer) programming solver based on the revised simplex method and the B&B method for the integers; it can be used as a library.). The tests were run on an Intel Xeon 1.6 GHz computer, with 4 GB of memory. Each run was stopped after a time out of 1200 s. We use the following measures to evaluate the performance of the solvers and of the various hybridisations:

- value of the first-found integer solution;
- CPU time (in s) to obtain the first-found integer solution;
- value of the best-found integer solution;
- CPU time (in s) to compute the best-found integer solution
- CPU time (in s) to prove optimality.

Test problem set  The solvers and their hybridisations will be tested with various instances of four different types of problems:

- the multiple balanced academic curriculum problem (mbacp),
- the set covering problem (scp),
- the set partitioning problem (spp)
- and the multidimensional knapsack problem (mknap).

The instances and model of MBACP can be found in Gent and Walsh (1999), whereas the instances and models of the other problems can be found in Beasley (1990). The size of the instances, in terms of variables and constraints, together with the best-known solution and the best-known CPU time are given in Table 1.

Table 1. Problem instances.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Number of variables</th>
<th>Number of constraints</th>
<th>Best-known solution</th>
<th>Best-known time</th>
</tr>
</thead>
<tbody>
<tr>
<td>mbacp1</td>
<td>572</td>
<td>473</td>
<td>1</td>
<td>0.81</td>
</tr>
<tr>
<td>mbacp2</td>
<td>714</td>
<td>656</td>
<td>0</td>
<td>8.44</td>
</tr>
<tr>
<td>mbacp3</td>
<td>856</td>
<td>346</td>
<td>–</td>
<td>3.05</td>
</tr>
<tr>
<td>mbacp12</td>
<td>717</td>
<td>570</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>mbacp13</td>
<td>859</td>
<td>578</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>mbacp23</td>
<td>859</td>
<td>650</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>mbacp123</td>
<td>862</td>
<td>882</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>scp65</td>
<td>1000</td>
<td>200</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>scpaa1</td>
<td>3000</td>
<td>200</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>scpb1</td>
<td>3000</td>
<td>300</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>sppaa01</td>
<td>8904</td>
<td>823</td>
<td>56,137</td>
<td>238.76</td>
</tr>
<tr>
<td>sppaa03</td>
<td>8627</td>
<td>825</td>
<td>49,649</td>
<td>12.47</td>
</tr>
<tr>
<td>sppaa04</td>
<td>7195</td>
<td>426</td>
<td>26,374</td>
<td>319.19</td>
</tr>
<tr>
<td>sppnw18</td>
<td>10,757</td>
<td>124</td>
<td>34,160</td>
<td>2.19</td>
</tr>
<tr>
<td>mknapcb2-1</td>
<td>250</td>
<td>5</td>
<td>59,312</td>
<td>–</td>
</tr>
<tr>
<td>mknapcb4-a</td>
<td>100</td>
<td>10</td>
<td>23,064</td>
<td>–</td>
</tr>
<tr>
<td>mknapcb4-b</td>
<td>100</td>
<td>10</td>
<td>22,801</td>
<td>–</td>
</tr>
<tr>
<td>mknapcb4-c</td>
<td>100</td>
<td>10</td>
<td>22,131</td>
<td>–</td>
</tr>
</tbody>
</table>
3.1. *lp_solve* Alone: a reference for the hybridisations

As a reference to compare the different hybridisations that will follow, Table 2 shows the results obtained with *lp_solve* alone. The first column shows the value of the first solution and the required time to compute it; the second column shows the same numbers for the best solution; the last column shows for the time required for proving optimality (‘–’ when optimality is not proven before the 1200 s of timeout).

With the same model, Gecode is faster to find a first solution, but of worse quality; it thus has to explore a larger search space to find the optimum which it never reached before the timeout. However, the reason is rather simple: we use the same linear model for *lp_solve* and Gecode, and this model is not well suited for Gecode. Since these results are not very relevant and do not show the abilities of Gecode, we do not include this table here.

3.2. Basic hybridisation

In this section, we do not consider any updating rule: propagation is triggered at each node of the search tree, and thus, the snapshots and indicators are not used (Table 3). This basic hybridisation should be seen not only as a reference for the next hybridisations that make use of our updating rules, but also as a strategy that has its own interest. Indeed, depending on the class of problems, each hybridisation leads to different improvements.

![Table 2. Results with *lp_solve*.](attachment:table2.png)
For example, the basic hybridisation is worth and efficient if one is interested in proving optimality of solutions in MBACP instances.

**MBACP** Times for proving optimality are significantly better than with the individual solvers (up to 29 times quicker for mbacp2). The cost of propagation is rather small, between 2% and 4% (depending on the MBACP instance) of the total execution time. However, the propagation is quite inefficient.

**SCP** The basic hybridisation worsens the CPU times to get both the first and best solution. The propagation cost is around 5%; however, the reason is that propagation is quite inefficient for these problems and thus stops quickly.

**SPP** Here, propagation is more efficient, but its cost is rather high between 26% and 40% of the total CPU time required for solving the instances. This extra cost penalises the basic hybridisation since it becomes slower than *lp_solve* alone.

**MKNAP** These are the most difficult problems for the solvers. It seems that numerous valid solutions must be evaluated and the search tree is much larger. The basic hybridisation is a bit faster than *lp_solve* to get the first solution, and to find the best solution in two cases over four.

### 3.3. Adaptive hybridisation based on fixed variables

This second hybrid reduces the number of propagation phases and hence tries to make them more efficient. The idea is to trigger propagation at some nodes only depending on

<table>
<thead>
<tr>
<th>Table 3. Results with the basic hybridisation.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic hybridisation</strong></td>
</tr>
<tr>
<td><strong>First solution</strong></td>
</tr>
<tr>
<td><strong>Value</strong></td>
</tr>
<tr>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>mbacp1</td>
</tr>
<tr>
<td>mbacp2</td>
</tr>
<tr>
<td>mbacp3</td>
</tr>
<tr>
<td>mbacp12</td>
</tr>
<tr>
<td>mbacp13</td>
</tr>
<tr>
<td>mbacp23</td>
</tr>
<tr>
<td>mbacp123</td>
</tr>
<tr>
<td>scp65</td>
</tr>
<tr>
<td>scp1</td>
</tr>
<tr>
<td>scp1</td>
</tr>
<tr>
<td>sppaa01</td>
</tr>
<tr>
<td>sppaa03</td>
</tr>
<tr>
<td>sppaa04</td>
</tr>
<tr>
<td>sppnw18</td>
</tr>
<tr>
<td>mknpcba4a</td>
</tr>
<tr>
<td>mknpcba4b</td>
</tr>
<tr>
<td>mknpcba4c</td>
</tr>
<tr>
<td>mknpcba2-1</td>
</tr>
</tbody>
</table>
the number of fixed (by enumeration or propagation) variables. To this end, we consider some percentage of fixed variables as shown in the following indicators.

3.3.1. Indicators

\( F \) is the snapshot taken at the father node and \( F^\prime \) at the grandfather node:

- \( \delta b_1 \) is a binary indicator which is true when at least one snapshot has been taken, false otherwise.
- \( \delta n_1 = (VF_F - VF_{F^\prime})/V_{\text{total}} \) represents the variation of the percentage of fixed variables between the father and grandfather nodes (\( V_{F^\prime} = 0 \) if there is no grandfather).

3.3.2. Updating rules

We consider the following rules for triggering the propagation or not:

\[
\begin{align*}
r_1: & \quad \text{not}(\delta b_1) \quad \rightarrow \quad \text{Propagation()}
\end{align*}
\]

\[
\begin{align*}
r_2: & \quad |\delta n_1| > P \quad \rightarrow \quad \text{Propagation()}
\end{align*}
\]

\[
\begin{align*}
r_3: & \quad [VF_F \times 100 \pmod{P \times 100}] = 0 \quad \rightarrow \quad \text{Propagation()}
\end{align*}
\]

The parameter \( P \) represents a minimum variation (in percentage) of fixed variables required to judge propagation as efficient. For our experimentations, parameter \( P \) was tried from 0% to 5%, with an increment of 0.1%. With \( P = 0 \) the propagation is achieved at each node of the search tree; we stopped with \( P = 5\% \) since for all the instances (of our test set) propagation was performed in less than 1% of the nodes with \( P = 5\% \): increasing more \( P \) does not make any relevant change.

Rule \( r_1 \) means that propagation must be done at the root node. Rule \( r_2 \) establishes that if the percentage of fixed variables increased of at least \( P \), then propagation should be triggered. Rule \( r_3 \) keeps propagation periodical (each \( P\% \) of fixed variables, whatever number of nodes are needed for it). When no rule applies, propagation is not triggered. Results are presented in Table 4.

3.3.3. Experiment

**MBACP** The percentage of fixed variables does not produce any significant change. Looking closer at the impact of \( P \) for finding the first solution for an MBACP problem, we discovered that the CPU time for finding the first solution is not really related to the variation of \( P \): while increasing \( P \) continuously, the time for the first solution oscillates from better to worse than with the basic hybridisation.

**SCP** For the SCP instances, not only the propagation cost is very low, but the effectiveness is also very low. We can observe that when \( P \) is large, the hybridisation behaves like \textit{lp\_solve} alone. With a small \( P \), it takes more time to obtain the best value and to prove optimality.

**SPP** It is not clear how propagation affects the results. For example, for the sppnw18 instance, some more observations showed us that increasing \( P \) from 0% to 0.4% significantly speeds up the resolution (from 42 s to 18 s). But increasing \( P \) over 0.4% does
not improve the solving process anymore. However, for the SPP instances, it is clear that with $P > 2$, the hybridisation behaves as $lp.solve$ alone.

**MKNAP** The improvement is to reduce the time to get the best solution and to get a good first solution of the quality of $lp.solve$ as quickly as possible.

### 3.4. Adaptive hybridisation based on propagation rate

The idea for this hybridisation comes from the observations in Figures 2–4, which show the average number of variables fixed by propagation with respect to the depth of the search tree. This is illustrated for the instances of the MBACP problem in Figure 2, of the SPP problem in Figure 3 and of the MKNAP problem in Figure 4. The $x$-axis represents the depth of the search (i.e. the depth of the currently explored node of the tree), and the $y$-axis the average number of variables fixed by propagation. The figures show the change of effectiveness of propagation (in terms of variables instantiated) during the search; indeed, depending on the solving state, propagation may be more or less useful.

In Figure 2, we can observe that for the case of the MBACP instances, propagation fixes numerous variables in the first level of the search tree (left part of the graphic, around 10 variables are instantiated by propagation on average); deeper in the search tree, propagation is not so effective (middle of the right part of the figure, one or two variables are instantiated during phases of propagation), even not at all after a while (the number of variables instantiated by propagation falls to 0).

For the SPP (Figure 3), this behaviour is even more obvious, especially in the first/upper levels of the search tree in which propagation fixes between 80 and more than 1000

---

**Table 4. Hybridisation triggering propagation w.r.t. the percentage of fixed variables.**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Best Value</th>
<th>First solution</th>
<th>Best solution</th>
<th>Optimality time</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>6.1</td>
<td>3</td>
<td>1</td>
<td>20.1</td>
</tr>
<tr>
<td>m2</td>
<td>5.2</td>
<td>3</td>
<td>0</td>
<td>10.8</td>
</tr>
<tr>
<td>m3</td>
<td>1.0</td>
<td>4</td>
<td>1</td>
<td>8.2</td>
</tr>
<tr>
<td>m12</td>
<td>8.8</td>
<td>4</td>
<td>1</td>
<td>85.8</td>
</tr>
<tr>
<td>m13</td>
<td>6.5</td>
<td>4</td>
<td>1</td>
<td>32.2</td>
</tr>
<tr>
<td>m23</td>
<td>5.4</td>
<td>4</td>
<td>1</td>
<td>68.5</td>
</tr>
<tr>
<td>m123</td>
<td>1.2</td>
<td>4</td>
<td>3</td>
<td>76.8</td>
</tr>
<tr>
<td>scp65</td>
<td>1.2</td>
<td>166</td>
<td>161</td>
<td>9.3</td>
</tr>
<tr>
<td>scp6a1</td>
<td>4.6</td>
<td>271</td>
<td>253</td>
<td>107.4</td>
</tr>
<tr>
<td>scp6b1</td>
<td>4.7</td>
<td>75</td>
<td>69</td>
<td>11.3</td>
</tr>
<tr>
<td>spp6a01</td>
<td>2.0</td>
<td>56,363</td>
<td>56,363</td>
<td>26.0</td>
</tr>
<tr>
<td>spp6a03</td>
<td>2.0</td>
<td>49,734</td>
<td>49,649</td>
<td>12.1</td>
</tr>
<tr>
<td>spp6a04</td>
<td>2.0</td>
<td>27,802</td>
<td>27,379</td>
<td>13.8</td>
</tr>
<tr>
<td>spp6n18</td>
<td>1.2</td>
<td>342,950</td>
<td>340,160</td>
<td>1.7</td>
</tr>
<tr>
<td>mkn6c4a</td>
<td>6.9</td>
<td>21,277</td>
<td>23,057</td>
<td>0.0</td>
</tr>
<tr>
<td>mkn6c4b</td>
<td>4.0</td>
<td>20,189</td>
<td>22,704</td>
<td>0.0</td>
</tr>
<tr>
<td>mkn6c4c</td>
<td>6.5</td>
<td>20,978</td>
<td>22,131</td>
<td>0.0</td>
</tr>
<tr>
<td>mkn6c2-1</td>
<td>5.6</td>
<td>57,318</td>
<td>59,013</td>
<td>0.1</td>
</tr>
</tbody>
</table>

---

**Note**

The values in the table represent the percentage of variables fixed by propagation.

---

**E. Monfroy et al.**

Downloaded by [UGR-BTCA Gral Universitaria] at 04:25 04 March 2015
Figure 2. Variables propagated w.r.t. the depth of the search tree: case of the MBACP.

Figure 3. Variables propagated w.r.t. the depth of the search tree: case of SPP.
variables (depending on the problem instance); then, deeper than 10 nodes, propagation starts to be useless for some instances (0 variables fixed by propagation) and less effective for the others (around 40 variables and then decreasing to 0).

The behaviour is a bit different for the MKNAP instances (Figure 4): propagation is not effective in the first levels of the search tree due to the fact that the instances are less constrained, and thus offer numerous possible solutions: we can see that on the left of the graphic, propagation does not fix any variable. But then, after some steps (reaching depth 11 of the search tree) and cutting the problem into subproblems, propagation becomes effective.

Thus, our idea is to achieve more propagation phases when we observe that propagation fixes variables and less propagation phases when propagation is less effective in terms of fixed variables. To consider propagation, we define the threshold \( \text{th} \) as the minimum number of variables fixed by propagation. In the snapshots, we have the total number of variables fixed by propagation and the number of nodes we have explored. Hence, we can get an indicator to reflect the efficiency of propagation at a given depth of the search tree.

3.4.1. **Indicators**

The following indicators enabled us to implement this idea. \( F \) is the snapshot taken at the father node. \( \text{snapshots}(h) \) represents the set of snapshots taken at depth \( h \).

- \( \delta n_1 = \#\text{snapshots}(\text{Depth}_F) \) represents the quantity of snapshots taken at depth \( \text{Depth}_F \) of the father node.
\[ \delta n_2 = \sum_{G \in \text{snapshots}} (\text{Depth}_G) (V_{\text{prop}})_G \] is the total number of variables fixed by propagation. Since we know the number of snapshots and the total number of variables fixed by propagation in each of these observations, we can determine the effectiveness of propagation at the depth of the father node.

### 3.4.2. Updating rules

The UPDATE component uses the following rules:

- \( r_1 : \quad \delta n_1 = 0 \quad \rightarrow \quad \text{Propagation}() \)
- \( r_2 : \quad (\delta n_2 / \delta n_1) \geq \text{th} \quad \rightarrow \quad \text{Propagation}() \)

Rule \( r_1 \) triggers propagation as long as we do not have observations that prove the inefficiency of propagation at this level. Rule \( r_2 \) evaluates the efficiency of propagation in the observed nodes; if this is greater than the parameter \( \text{th} \), then propagation is triggered. If no rule applies, then propagation is deactivated.

### 3.4.3. Experiments

To test this hybridisation (see Table 5), the parameter \( \text{th} \) will vary from 0 to 20: with \( \text{th} = 0 \), propagation is triggered at each node of the search tree; experimentally, we observed that if \( \text{th} > 20 \) then propagation is never triggered. The results are presented in Table 5.

**MBACP** For the larger instances (i.e. m12, m13 y m123) this hybridisation improves the results compared to the basic hybridisation. With a low threshold, we could shorten CPU
times from 30% to 40%. As observed initially in the basic hybridisation, propagation is fast but rather inefficient for these instances: thus, to trigger propagation, a low threshold is required.

**SCP** In these instances, there is no improvement for proving optimality: the timings are similar to the ones of the basic hybridisation, and the ‘best’ threshold th produces the same results as *lp_solve* alone. This confirms the idea that propagation does not improve this type of problems and just slows down the solving process.

**SPP** Here, the results are better than for the basic hybridisation, but a little bit worse than *lp_solve* alone. Except for the instance sppnw18, for which this hybridisation improves time for optimality w.r.t. to all the other solvers (hybrid or not). For this type of problems, although propagation effectively reduces the search space and the search tree, the overhead is higher than the speed-up for the global solving process.

**MKNAP** On average, this hybridisation is a bit faster than the best-tested solver (*lp_solve* for these instances). It also gives some good quality solutions. After a closer look at the mknapcb4b instance, we can tune the threshold in order to get better solutions than with *lp_solve*; however, the cost of these better solutions is to perform propagation 40% of the total time.

### 3.5. Adaptive hybridisation based on a distance in the search tree

We now want to penalise propagation when we judge it is inefficient. To this end, the hybridisation starts performing propagation; the solving process carries on with propagation as long as the propagation phase brings relevant information. If at a given node the propagation does not fix any variable, then the propagation phase is not performed in the following next *d* nodes, *d* being a distance (i.e. a number of nodes) we fix *a priori*. The larger the value for *d*, the less propagation phases will be achieved.

#### 3.5.1. Indicators

The indicators only use the last taken snapshot (*F*):

- **δb_1**: this indicator is true if at least one snapshot was taken; it is false otherwise.
- **δn_1** = \((V_{f_{prop}})_F\) is the number of variables fixed by the last propagation phase.

#### 3.5.2. Updating rules

The penalty *dist* represents the remaining distance before applying propagation again: it is fixed to *d* when propagation is penalised, and then decreases of 1 at each following node. The following rules are used in the UPDATE component:

\[
\begin{align*}
\ r_1: & \quad \neg(\delta b_1) \lor \text{dist} = 0 \lor \delta n_2 > 0 \quad \rightarrow \quad \text{dist} = 0; \ \text{propagation()} \\
\ r_2: & \quad \delta n_2 = 0 \land \text{dist} = 0 \quad \rightarrow \quad \text{dist} = d; \ \text{nopropagation()} \\
\ r_3: & \quad 0 < \text{dist} < d \quad \rightarrow \quad \text{dist} = \text{dist} - 1; \ \text{nopropagation}().
\end{align*}
\]

Rule \(r_1\) means that the propagation phase must be triggered if no snapshot has been taken (beginning of the solving process) or propagation fixed variables in the father node, or \(\text{dist} = 0\) (meaning that propagation is not currently penalised). The other rules apply
when propagation does not fix variable. Rule $r_2$ penalises propagation with the distance $d$ and Rule $r_3$ decreases the penalty of the already penalised propagation.

3.5.3. Experiments

We made some experimentations (Table 6) with the distance $d$ ranging from 0 to 20 nodes, with an increment of 1. With $d=0$, this hybridisation is equivalent to the basic hybridisation. With $d=20$, propagation is nearly never used.

**MBACP** This strategy improves the results of the basic hybridisation and of *lp_solve* in various cases. In three instances, the best parameter $d$ enables to reduce 30% of the time needed for proving optimality w.r.t. the basic hybridisation.

**SCP** For these instances, increasing $d$ leads to results similar to *lp_solve*. However, with $d$ larger than 6 there is no more improvement.

**SPP** For proving optimality, there is no clear tendency: for sppaa03, increasing the distance we reach the efficiency of *lp_solve*; but, performing propagation and observing the solving process, we degrade the performance compared to *lp_solve*. For sppnw18, with $4 \leq d \leq 8$ the results are better than *lp_solve*.

**MKNAP** On average, this hybridisation is faster than *lp_solve* for finding the same ‘best’ solution. However, for proving optimality of mknapc4c, we see that propagation is an overhead. We could observe that using a larger $d$, we obtained better results than *lp_solve*, and thus, propagation effectively benefits to this type of problems. However, tuning correctly $d$ for all these instances is rather complicated.
3.6. Adaptive hybridisation based on CPU time used by propagation and relaxation

This strategy is based on the observations of the CPU times required for relaxation, propagation and relaxation after propagation w.r.t. the current depth in the search tree. The idea of this strategy came from some detailed observations we made with some instances. In proportion to the depth, the effect of propagation is to provide subproblems that are smaller, and that relaxation can solve quickly. However, propagation time must be taken into account w.r.t. the shorter time of relaxation. Our goal is thus to take CPU time into account in order to avoid unnecessary propagation phases or phases that last longer than the benefit for relaxation.

This strategy and the indicators use the durations of propagation and relaxation, and two groups of snapshots: $\text{snapshots}_1$ (respectively $\text{snapshots}_2$) the snapshots taken at depth $\text{Depth}(F)$ for nodes that were treated (respectively that were not treated) by propagation. $F$ is the father node:

### 3.6.1. Indicators

- $\delta_n_1 = \#\text{snapshots}_1$ (respectively, $\delta_n_2 = \#\text{snapshots}_2$) represents the number of snapshots for nodes that got (respectively, did not get) propagation.
- $\delta n_3 = (\sum S_{\text{snapshots}_1}(T_{\text{prop}}))/\delta n_1$ represents the average time of propagation phases for snapshots of $\text{snapshots}_1$.
- $\delta n_4 = (\sum S_{\text{snapshots}_1}(T_{\text{relax}}))/\delta n_1$ represents the average time of the relaxation phases that happened after propagation.
- $\delta n_5 = (\sum S_{\text{snapshots}_2}(T_{\text{relax}}))/\delta n_2$ represents the average time of the relaxation phases of nodes that were not treated by propagation.

### 3.6.2. Updating rules

The following rules are used to manage the strategy:

- $r_1$: $\delta n_1 = 0$ \quad $\rightarrow$ Propagation()
- $r_2$: $\delta n_2 = 0 \land \delta n_1 \neq 0$ \quad $\rightarrow$ NoPropagation()
- $r_3$: $\delta n_1 > 0 \land \delta n_2 > 0 \land (\delta n_3 + \delta n_4 > \delta n_5)$ \quad $\rightarrow$ Propagation()
- $r_4$: $\delta n_1 > 0 \land \delta n_2 > 0 \land (\delta n_3 + \delta n_4 < \delta n_5)$ \quad $\rightarrow$ NoPropagation().

Rule $r_1$ triggers propagation when there has not yet occurred a propagation phase at this level of the tree search. Rule $r_2$ deactivates propagation in the opposite condition. Rules $r_3$ and $r_4$ are the central idea of this hybridisation strategy: if there are some observations with and without propagation ($\delta n_1 > 0 \land \delta n_2 > 0$), we evaluate whether the treatment of the observed nodes takes less time with propagation or not; in the positive case, the propagation phase is triggered by Rule $r_3$, otherwise propagation is deactivated by Rule $r_4$.

### 3.6.3. Experiments

**MBACP** As we observed earlier, in Table 7 we can see that the combined CPU times of relaxation and propagation in the hybridisation are smaller than just with relaxation, and this, for all the instances. The table shows that this hybridisation obtained good results; the main benefits are (1) the time reduction to obtain the best solution and (2) an improvement of the objective w.r.t. $lp\_solve$ for the instances for which we could not prove optimality before.
SCP  For all the instances, the timings are a little bit worse than \textit{lp\_solve}; however, they are still acceptable since the overhead is not so much.

SPP  This hybridisation improves the best solution of the sppaa04 instance w.r.t. \textit{lp\_solve}. However, this improvement costs 8 times the CPU time required by \textit{lp\_solve} to get its best solution, and it is 4 times slower to get the same solution as \textit{lp\_solve}: if one is only interested in solution quality, this hybridisation is interesting, but for someone trying to get a trade-off between quality and time, this hybrid is not really a candidate. For sppnw18, only half of the CPU time of the basic hybridisation is needed to prove optimality.

MKNAP  Although optimality is not proved for mknapcb4a, a solution nearly as good as the best one of \textit{lp\_solve} is obtained in less than half the time needed for \textit{lp\_solve}. As noted before for this problem class, early propagation is significantly beneficial; however, this benefit disappears if propagation is always performed, and the results can even become worse than without propagation. In this case, looking closer at the solving process, propagation is triggered at the beginning, but then, it is not triggered again. Whereas for mknapcb4b and mknapcb4c, the same best solution is obtained a bit slower and for mknape2-1 the CPU time is improved.

4. Discussion, conclusion and future work

4.1. Discussion

Comparing our framework with related works is not so simple. First, most of these studies are systems that implement one hybridisation strategy for a given class of problems.

<table>
<thead>
<tr>
<th></th>
<th>Best solution</th>
<th>Best solution</th>
<th>Optimality time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Value</td>
<td>Time</td>
<td>Value</td>
</tr>
<tr>
<td>m1</td>
<td>3</td>
<td>2.4</td>
<td>1</td>
</tr>
<tr>
<td>m2</td>
<td>4</td>
<td>3.1</td>
<td>0</td>
</tr>
<tr>
<td>m3</td>
<td>4</td>
<td>1.0</td>
<td>1</td>
</tr>
<tr>
<td>m12</td>
<td>4</td>
<td>11.1</td>
<td>1</td>
</tr>
<tr>
<td>m13</td>
<td>4</td>
<td>3.1</td>
<td>1</td>
</tr>
<tr>
<td>m23</td>
<td>4</td>
<td>2.3</td>
<td>1</td>
</tr>
<tr>
<td>m123</td>
<td>4</td>
<td>103.0</td>
<td>3</td>
</tr>
<tr>
<td>scp65</td>
<td>166</td>
<td>0.2</td>
<td>161</td>
</tr>
<tr>
<td>scp1</td>
<td>271</td>
<td>0.9</td>
<td>253</td>
</tr>
<tr>
<td>scp1</td>
<td>75</td>
<td>1.2</td>
<td>69</td>
</tr>
<tr>
<td>sppaa01</td>
<td>56,363</td>
<td>33.5</td>
<td>56,363</td>
</tr>
<tr>
<td>sppaa03</td>
<td>49,734</td>
<td>17.4</td>
<td>49,649</td>
</tr>
<tr>
<td>sppaa04</td>
<td>27,802</td>
<td>17.1</td>
<td>27,412</td>
</tr>
<tr>
<td>sppnw18</td>
<td>342,950</td>
<td>1.8</td>
<td>340,160</td>
</tr>
<tr>
<td>mknape,b4a</td>
<td>21,277</td>
<td>0.0</td>
<td>23,050</td>
</tr>
<tr>
<td>mknape,b4b</td>
<td>20,189</td>
<td>0.0</td>
<td>22,704</td>
</tr>
<tr>
<td>mknape,b4c</td>
<td>20,978</td>
<td>0.0</td>
<td>22,131</td>
</tr>
<tr>
<td>mknape2-1</td>
<td>57,318</td>
<td>0.1</td>
<td>59,013</td>
</tr>
</tbody>
</table>
Our framework is more general since it enables one to manage solver strategies (shown in Castro et al. 2005), jumps back in the search tree (shown in Monfroy et al. 2006), and functions of solvers and hybridisation strategies (shown in this article). Furthermore, it is not specialised for a given class of problems, but it is flexible and open: for example, in Crawford, Castro, and Monfroy (2010), the UPDATE component is changed in order to manage enumeration strategies with hyper heuristic instead of priorities.

We now compare with a selection of approaches that are related to our method and try to enlighten the differences. In Rodosek et al. (1999), the fixed hybridisation strategy is similar to the basic hybridisation of Section 3.2 with a mechanism to discover reified and all different constraints that are also used by the solving process.


Lobjois and Lemaitre (1998) proposes a method called Selection by Performance Prediction which allows one, when faced with a particular problem instance, to select a B&B algorithm from several promising ones to solve a particular problem instance. In this case, the selection is made a priori and the solver does not adapt itself dynamically.

4.2. Conclusion

In Castro et al. (2005) a framework for adaptive enumeration strategies and metabacktracks for a propagation-based constraint solver has been studied. In this article, we extended this framework in order to trigger some functions of a solver, or of a hybrid solver. We then presented how simple adaptive hybridisation strategies can be designed with this extension by just changing few rules that react to the behaviour of the solving process.

We experimented this framework on a hybrid B&B + propagation solver in which propagation can be triggered to respond to some observations of the solving process. The results show that not only some phases of propagation are beneficial to the B&B algorithm, but also that propagation is too costly to be executed at each node of the search tree. The hybridisation strategies are thus crucial in order to ‘tune’ when to perform the or not propagation. Depending on the type of problems, we have also seen that different hybridisations lead to different improvements, i.e. none of the hybridisation is always the best. This is of course related to the No Free Lunch theorems for optimisation (Wolpert and Macready 1997) which argue against the existence of a universal multipurpose solver. The challenge is thus to obtain more autonomous and adaptive solving algorithms in order to treat efficiently larger classes of problems.

4.3. Future work

We thus plan to integrate a learning module that would enable to select the hybridisation with respect to the problem to be solved. The next step would be to change dynamically the hybridisation strategy in order to respond to the change of the search state. This would lead to a kind of meta level of our framework which would consist in:

- changing strategies and triggering functions adaptively using some rules.
- and also adaptively change these rules or set of rules.
Acknowledgements
This work was partially funded by the TIC Amsud project TODAS, the INRIA-CONICYT project CoreWeb, the INRIA Associated Team WanaWeb, and the Centro Cientifico-Tecnologico de Valparaiso (CCTVal), FB0821.

References
