Search Techniques for CSPs with Ordered Domains

Technical Report No. TR-99/313

Marius-Călin Silaghi, Djamila Sam-Haroud, and Boi Faltings

Laboratoire d'Intelligence Artificielle
Département d'Informatique
EPFL, Ecublens
CH-1015 Lausanne

E-mail : { silaghi / haroud } @lia.di.epfl.ch

Fax: ++41-21-693.52.25

May, 1999

This report presents an approach for searching in CSPs with ordered domains. It combines intelligent domain splitting with backtracking to find the whole solution space. Two algorithms are presented. The first one, called Box-DCPR, implements a look-ahead strategy while the second, called COB, is an intelligent backtracker. Both algorithms can be applied to general systems of constraints with explicit representations.

x/y	a	b	c	d
e	1	0	1	0
f	0	1	0	1
g	1	0	1	0
h	0	1	0	1

(1)

x/y	a	c	b	d
e	1	1	0	0
g	1	1	0	0
h	0	0	1	1
f	0	0	1	1

(2)

Figure 1: *A discrete constraint between two variables x and y. x takes its values in the set $\{e,f,g,h\}$ and y in $\{a,b,c,d\}$. Ordering the values according to (2) induces a more important grouping of the feasible values within the constraint representation than ordering them according to (1).*

1 Introduction

Determining the complete solution set of a constraint satisfaction problem is important in many applications such as spreadsheet computation, design with multi-criteria optimization or negotiation. The task of characterizing all solutions for a CSP is more difficult than that of generating the first solution. It is confronted to complexity problems for both extracting and representing the solution space. An intuitive way around this complexity barrier is to structure the search space so that the exploration algorithm operates on aggregated subsets of data rather than on individual possible instantiations. In the discrete case, the cross-product representation (CPR) proposed by [10] is a powerful technique for compactly describing the complete solution space of a CSP. The idea consists of combining several partial solutions into a union of cross products, each cross-product representing an aggregated set of values. In the case of continuous domains, aggregated sets of values are typically represented using intervals. The method we propose combines the strengths of the two aggregation approaches for ordered discrete CSPs.

The interval representation naturally extends to the discrete case for ordered domains. Interval-based search and local propagation techniques from continuous domains can henceforth be transfered to discrete ordered CSPs. Numerical search techniques based on intervals are essentially dichotomic. Variables are instantiated using intervals. When the search reaches an interval that contains only solutions it stops, otherwise the interval is recursively split into smaller sub-intervals up to an established resolution. The whole process is generally interleaved with local consistency propagation to prevent combinatorial explosion [11].

While, in classical numerical search with intensional representation of constraints it is more practical to use a rigid binary split mechanism, the explicit constraint representation we use supports the implementation of a flexible and more intelligent splitting technique.

The new lookahead backtracking algorithm we present, called Box-DCPR, combines intelligent splitting with a local consistency technique inspired from box-consistency [8], one of the most powerful propagation algorithms in continuous domains. The solution space is generated concisely using CPR. We will show that the interval representation greatly simplifies the implementation of CPR in ordered domains. We also present a no-good based backtracking algorithm for ordered domains that incorporates value aggregation. This algorithm is called COB.

The condition of having ordered domains is fulfilled by most constraint types [3]. The order can be arbitrary, heuristically determined or natural, as in numerical applications. Each ordering of the values in the domains induces a certain grouping of feasible values within the

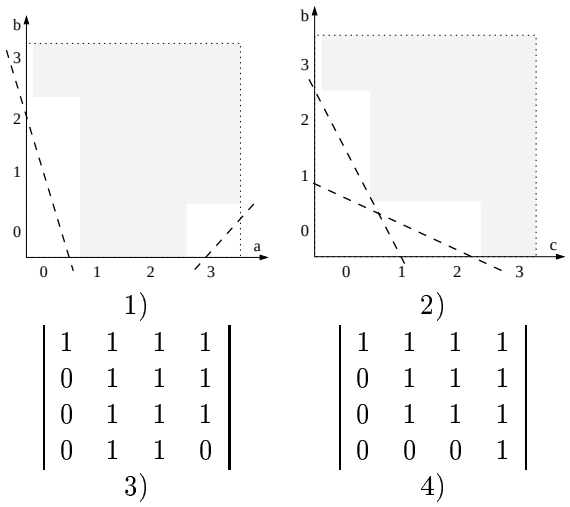


Figure 2: *An integer numerical CSP with three variables a, b, c of domain {0,1,2,3} and four linear inequalities (dashed lines). 1)the constraints between a and b are* $b \geq 3 - 3a \wedge b > a - 3$*; 2) the constraints between b and c are* $b \geq 3 - 2a \wedge 2b \geq 3 - a$*; 3)the (0,1)-matrix representation of the first constraint; 4) the (0,1)-matrix representation of the second constraint.*

explicit constraints representations (Figure 1). We characterize a given order by a measure called its density, computed as the ratio between the number of feasible neighborhoods and the number of feasible elements in the explicit (0,1)-matrices representation of the constraints. We expect our approach to be at its best for the applications where the order is given and the density naturally high, as it is often the case for underconstrained numerical problems in finite or discretized domains, scheduling and graph coloring problems etc.

2 Example

We start by illustrating the notion of intelligent split on a small example.

Let us consider the integer numerical CSP of Figure 2. The problem has three variables a, b and c, each of which takes its values within the ordered set {0,1,2,3}. The variables a and b are constrained by the conjunction of two inequalities, and so are also b and c (Figure 2(1) and (2)). We assume that the constraints are represented using the discrete $(0, 1)$-matrix representation and that the domains are ordered according to the natural order of the integers.

A standard backtracking algorithm for discrete CSPs would require 64 checks to generate all solutions of this CSP: each of the four possible values for c must be tested against the 12 feasible pairs of values for a and b. Determining the 12 feasible pairs for a and b requires a total of 16 checks.

Assume now that rather than assigning to a given variable an individual value at each step, we would like to assign simultaneously a group of values.

A first way to proceed would be to perform a dichotomic search, as described for continuous CSPs. Since the domains are ordered, an aggregation of values can be represented using an interval and the binary split schema applies. Using this technique, the solution region determined by the constraints between a and b would be decomposed into six regions, as shown in Figure 3(1).

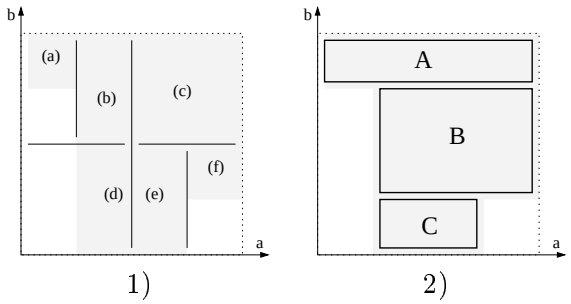


Figure 3: *Approaches to domain splitting: 1) Binary split; 2) Intelligent split.*

The number of checks required for generating all solutions is 56 in this case: 16 checks are necessary to decompose the constraints into 6 feasible regions. Choosing a and b within region (a) or (f) implies 4 checks per region for c and within region (b), (c), (d) or (e) 8 checks per region.

The binary split procedure implicitly aggregates values that can be assigned simultaneously. This allows for generating a more compact description of the solution space with a lower number of checks. The aggregation mechanism offered by dichotomic search remains however uninformed since the constraints descriptions are not taken into account for performing the split. Binary split has therefore great chance to cut just in the middle of feasible regions, requiring further useless splits to take place.

An intelligent splitting mechanism would try to reduce the number of feasible regions generated, by choosing appropriate splitting points. The solution region determined by the constraints between a and b could for example be decomposed into three regions (A), (B) and (C), as shown in Figure 3(2). The number of checks required for generating all solution reduces to 32 checks in that case: 16 checks must be used for generating the decomposition. Choosing a and b within region (A) or (C) implies 4 checks per region for c and within region (B), 8 checks.

In Section 4.2, we present an algorithm for achieving such kind of aggregations. The algorithm builds the regions by expansion on directions chosen using ordering heuristics. Although not optimal, the splitting strategy proposed significantly reduces the width of the search, as reported in the experiments.

3 Background

We consider the general case of n-ary CSPs and use their classical graph/hyper-graph representations. In the *primal* representation, the vertices represent the variables and the edges the constraints between them. In the *dual* one, a vertex stands for a constraint and the edges are equality constraints between the common variables [1].

We first propose a hybrid look-ahead backtracking algorithm that uses both the primal and the dual graph representations. The primal graph supports a compact formulation of the solution space. The dual one is used to perform the intelligent split mechanism and control the backtrack procedure. The algorithm we present is a variant of MAC (Maintaining Arc Consistency) [15]. MAC enforces arc-consistency on the remaining uninstantiated variables, after the instantiation of each variable. It has proven to be one of the best techniques for hard problems [4]. In our

approach, we maintain a weaker form of arc-consistency borrowed from continuous domains, namely box-consistency [8]. Box-consistency enforces arc-consistency on the outer bounds of a variable's domain. It produces a convex enclosure of the set of possible values.

In 1993, Ginsberg has published his Dynamic Backtracking (DB) where additivity on the size of disjoint subproblems for finding the first solution was guaranteed. The approach needs only polynomial space while the hitherto approaches, that had achieved this using no-goods, needed an exponential one. In [2] it was shown that for non-additive problems, the new scheme may decrease performance not only due to computational overhead, but also due to inflexibility in changing the variable ordering. The constraints were relaxed in the General Dynamic Backtracking (GDB) [14], with the risk of loosing the additivity property. In fact, random reordering allowed by GDB, may involve looses in no-goods for a subproblem, due only to reorderings occurring at backtrack in a different disjoint subproblem. A strong flexibility in the variable ordering, that preserves additivity, is offered by a new family of backtracking algorithms based on total assignments and partial order. The new algorithms were called Partial Order Backtracking (POB) [14] and Partial order Dynamic Backtracking (PDB) [7]. DB was also given some more flexibility by its redefinition within the new framework [7]. Recently, a generalization of the previous algorithms, that combines their advantages, was published as General Partial order Backtracking (GPB) [5]. The GPB algorithm keeps a total assignment and allows both the reordering of DB and PDB. An interesting feature of POB remains the ability of this algorithm to erase no-goods only after making sure that an antecedent variable is changed. This behavior can be included with no effort in GPB, as we present in the next section, and avoid erasing no-goods that might be reused immediately.

When a variable has to be chosen, the heuristics we have tested are the least domain (LD) variable [6] and the most constraining variable. LD proposes the variable with the smallest number of values as an easy to compute measure. LD is itself a heuristic for choosing the most constraining variable.

4 Algorithms

As mentioned before, the search technique we present builds on both the primal and the dual representations of CSPs. We choose to use the terms "variable" and "constraint" uniquely for the elements of the primal representation.

In the primal representation, the label assigned to a node is a discrete interval.

Definition 1 *Let I be a set of values taken from a totally ordered discrete domain D. I is a* discrete interval *if its elements are successive in D.*

A discrete interval can be represented extensively by listing all its elements, or intensively using its bounds.

In the dual representation, the domain for each node is a constraint. The nodes are labeled using the notion of multi-dimensional boxes, called multi-boxes.

Definition 2 *Let C be a constraint between a set $V = \{v_1, \ldots v_k\}$ of variables. Let $\{I_1, \ldots I_k\}$ be a set of discrete intervals respectively assigned to the elements of V from their domains. The cross product $I_1 \times \ldots I_k$ is a multi-box of C iff it satisfies C.*

The search technique basically proceeds as follows. The domain of each constraint is set to the cross-product of the domains of its variables when the constraint is selected for instantiation. A dual node c is selected and instantiated using a multi-box within its current domain. The label of the corresponding primal nodes are then restricted to the values allowed by the multi-box. This restriction is further propagated in the primal graph using a local consistency algorithm. If one of the variable domains is emptied, another multi-box is chosen for c. A backtracking is initiated if all multi-boxes of c have been exhausted. When all the dual nodes have been instantiated, a solution region is provided. This solution region is given by the cross-product between the interval labels of the primal graph. The use of the dual representation is mainly intended to enable an easy aggregation mechanism. We motivate this choice in more details in Section 5. We are now in position to present our algorithm. We start by describing the local propagation algorithm used during search.

4.1 Discrete Box Consistency

By enforcing arc-consistency on the outer bounds of the domains only, box-consistency spares a significant amount of work and space. It has proven particularly useful in continuous domains and is integrated in the most successful techniques for numerical CSPs. The use of this notion has also been suggested for problems in finite domains [13].

The extension of box-consistency from numerical to general ordered domains is natural and motivated by the same considerations. In the definition of discrete box-consistency, we simply replace the notion of continuous intervals by that of discrete ones:

Definition 3 *Let $\mathcal{S}$ be a system of discrete constraints with n variables $v_1, ..., v_n$ and $\mathcal{I}$ be a cross-product of n* discrete intervals *$\langle I_1, ..., I_n \rangle$, and let $l_i = min(I_i)$ and $u_i = max(I_i)$ $(1 \leq i \leq n)$. Let $C(\mathcal{I})$ represent the tuples of the cross-product $\mathcal{I}$ allowed by the constraint C. $\mathcal{S}$ is (discrete) box consistent in $\mathcal{I}$ wrt i if*

$$\{l_i, u_i\} \in \bigcap_{C \in \mathcal{S}} \prod_{v_i} C(\mathcal{I}).$$

$\mathcal{S}$ is (discrete) box-consistent in $\mathcal{I}$ if it is (discrete) box-consistent in $\mathcal{I}$ wrt all i in 1...n.

The algorithm 1 enforces discrete box-consistency. The implementation used is based on AC-6 [3]. It requires storing for each boundary of each checked constraint the tuple in the constraint that supports the boundary. This requires a space of $O(NC * MA^2)$.

```
function DBoxConsistency(Queue)
    while c=ExtractConstraint(Queue) do
        if !CheckBounds(Queue,c) then
            return 0
        end
    end
    return 1
end.
```

Algorithm 1: *Discrete Box-Consistency*

```
   function CheckBounds(Q,c)
     for all VariablesInConstraint(v,c) do
       changed←0
2.1    if OutOfBound(UpSupport(v,c)) then
         Box←ScanForUpper(v,c)
         changed← 1
       end
2.2    if OutOfBound(LowSupport(v,c)) then
         Box←ScanForLower(v,c)
         changed← 1
       end
       if Empty(Box,v) then
         return 0
       end
       if changed then
         Insert(Q,AffectedConstraints({v}))
2.3      UpdateVariableRange(Box,v)
       end
     end
     return 1
   end.
```

Algorithm 2: *Check Bounds*

Initially all the constraints are inserted in the queue. A constraint is queued when the domain of one of its variables changes. A revision step consists of verifying that the values supporting the boundaries of the current constraint are still valid. For any support falling outside the current intervals of the variables (lines 2.1 and 2.2), the constraint is scanned so that a new support can be identified. If the scanning is performed according to a static order of the domains and of the constraints' variables, then it is only needed to scan beyond the old support.

Theorem 1 *Discrete Box-Consistency terminates in less than $NC * DS * NV$ revision steps where DS and NV stand for the maximum domain size, respectively the number of variables.*

Proof. After each reduction of a domain there will be at most $NC - 1$ constraints checked without a subsequent domain reduction. If, after checking all the $NC - 1$ constraints implied, there is no domain reduction, consistency is reached and the algorithm stops. There are at most DS*NV domain reductions possible. Therefore there will be maximum NC*DS*NV steps before failure or consistency is reached. □

Theorem 2 *The complexity of the Discrete Box-Consistency algorithm is $O(NC * DS * (NV * MA + DS^{MA-1}))$ where MA stands for the maximum arity of a constraint.*

Proof. The domains of the constraints are scanned only once, this results in a number of $O(NC * DS^{MA})$ checks. The only values that have to be reconsidered as supports are those of the changed variables. However, at each revision step, all the supports of a constraint boundaries

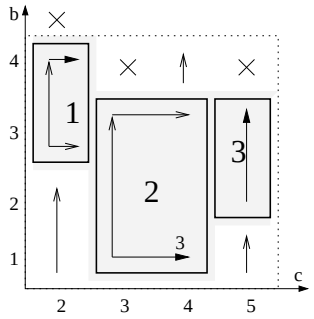


Figure 4: *Example of a run of the dynamic aggregation procedure. The arrows show the direction of the scanning. The crosses show the continuation points. The rectangles are the resulting multi-boxes and the big numbers inside show the order in which they are returned. Smaller number in the rectangles show the jump displacement for the scan procedure.*

(eventually one less[1]) have to be tested, resulting in a complexity of $O(NC * DS * NV * MA)$. □

One difference between discrete-box consistency and a standard AC algorithm, is that the modification of a domain D_k requires the constraints to be tested also for the reduction of the already reduced bounds of D_k. Such a situation is not possible in AC where the corresponding values in the reduced D_k have been already tested in previous passes.

4.2 Implementing the Aggregation

We now present the dynamic aggregation procedure we propose for computing the multi-boxes. This procedure scans a constraint according to an ordering of the variables. This ordering specifies priorities on the directions of aggregation. A progress information structure is used that stores the starting point for aggregation at each step. Given an order of the variables, at each request for a multi-box, we systematically scan the constraint up to a feasible tuple. The starting point is inherited from the previous call. Once a feasible tuple is found, we try to expand it systematically along all the variables according to the ordering provided by heuristics described section 6. The tuples of the box obtained are marked as already covered, so that future scanning and aggregations will skip them. The starting point for the next request is set to the right side of the box, on the direction with the highest priority at scanning. The disadvantage of this approach lies in the fact that it requires checking each time the tuples one by one. A risk of overhead exists, especially in problems with large domains, when we try to expand on other directions than the one with the highest priority at scanning. For the scanning procedure to skip wide covered boxes we assign to the left side tuples, the width of the box covered (jump displacement). Finally, it is most probable that a box can be aggregated more on its highest priority direction, which is the same as the highest priority scanning direction and this offers no overhead.

The aggregation procedure is illustrated by the example of Figure 4. The current domain of the constraint (c, b) is set to the cross-product $[2, 5] \times [1, 4]$. The iterator is set to the position

[1] If only one bound of only one variable is reduced, then the support of the other bound of the same variable don't have to be checked.

(2,1). Let us consider that the ordering heuristic gives us the order c, b. We scan the constraint as shown by the arrow starting at (2,1) and we discover a feasible tuple at (2,3). We extend it first for c obtaining the first box $[2,2]\times[3,4]$. Then we try to extend it for b, operation that fails because of the infeasible tuple (3,4). The continuation point is set to (2,5), and the multi-box 1 is returned.

When a new query for a box is received, the continuation point is read and the scan continues from (3,1), where it also finds a feasible tuple and stops. The obtained box is stretched first for c to the box $[3,3]\times[1,3]$. Then, it is expanded for b and it becomes $[3,4]\times[1,3]$. We do not try to expand it in the descending direction of a variable. The new continuation point is set to (3,4) and the jump displacement for scan in (4,1) to 3. Then the multi-box 2 is returned. In the same manner, at the third request, the scan begins from (3,4), reads (4,1) and jumps to (4,4). It continues on (5,1) and discovers a feasible tuple at (5,2). The box is expanded for c and the multi-box 3 is delivered. At the fourth request, the continuation point (5,4) is checked and then empty constraint is signaled. For certain constraints it might be the case that the box expansions disconnect the feasible regions and lead to aggregation on less efficient directions. For such constraints, additional checks at the borders of the boxes can determine the expansions with a less probable utility.

4.3 Box-DCPR

We describe in this section our maintaining box-consistency algorithm called Box-DCPR. At the beginning, the labels of all variables are set to their whole initial domains. Discrete box-consistency is called with all the constraints in the queue in order to set box-consistent labels. The search is started by calling Box-DCPR(0, nil). At step i, the constraints i is instantiated to a completely feasible multi-box, within the ranges accepted by the current labels of the variables. Each time a new constraint is entered, we estimate dynamically an order of the variables (of the directions of aggregation) of that constraint (line 3.5).

The instantiation is given by an iterator initialized at line 3.4 and that builds the multi-boxes by aggregation 4.2. The algorithm 4 performs the aggregation along all the directions with a computed priority given by a heuristic discussed in section 6 and is called at line 3.6. Between two calls, the last checked tuple in the scanning order that leaves no unchecked tuple behind, is stored as continuation point (line 4.3). If such an instantiation is successful, we intersect the current labels of the variables implied in the constraint with the ranges covered by the chosen multi-box (at line 3.8). If a given step alters the domains, then they have to be refreshed from the Stack (at line 3.7).

The Stack of Domains is pushed and popped each time a constraint is entered respectively left (line 3.3 respectively line 3.12). The instantiation of a constraint corresponds to a split of the constraint and several splits of variables. We perform a backtracking based on such steps. Whenever all the constraints are instantiated 3.1, we generate the solutions as the cross-product of the current labels of the variables (line 3.2).

The line 3.9 calls the algorithm achieving discrete box-consistency. We know that the instantiated constraints have supports for all their bounds and only the future constraints are revised. If the function fails, a new step begins with the search and aggregation of a new multi-box. At this step and at backtracking, the bounds supports used by discrete box-consistency will be restored. If the local propagation does not result in domain wipe out, the future constraints are reordered (line 3.10) based on the LD heuristic and then Box-DCPR is recursively called at

line 3.11.

```
    function Box-DCPR (i, Stack)
3.1 |  if LastConstraint(i) then
3.2 |  |  Report(SolutionCrossProduct)
    |  |  return
    |  end
3.3 |  PushCurrentDomains(Stack)
3.4 |  ResetIterator
3.5 |  ReOrderVariables(i)
3.6 |  while GetNextBox(i) do
3.7 |  |  LoadDomains(Stack)
3.8 |  |  UpdateDomainsAndConflicts(ChosenBox)
    |  |  Q←AffectedConstraints(i)
3.9 |  |  if !DBoxConsistency(Q) then
    |  |  |  continue
    |  |  end
3.10|  |  ReOrderConstraints(i)
3.11|  |  Box-DCPR(i + 1, Stack)
    |  end
3.12|  PopCurrentDomains(Stack)
    |  return
    end.
```

Algorithm 3: *Search BOX-DCPR*

Theorem 3 *The algorithm Box-DCPR is correct and complete.*

Proof. Each solution tuple has at least one corresponding feasible sub-tuple in each constraint. When a solution is generated as the cross-product of the variables labels, we know that all the constraints have been taken into account and instantiated with multi-boxes (cross-products). It is guaranteed by construction that the projection of any multi-box over a given variable results into an interval that contains the label of that variable. This guarantees correctness. The aggregation algorithm spans exhaustively the entire space of the constraint compatible with the instantiation of the past constraints. The box consistency is obviously complete. Therefore the Box-DCPR is complete. □

4.4 General Partial Order Backtracking (GOB)

For a set X of N variables taking their values in the domains D_k, $k = \overline{1..N}$, the inference rules [14] used in no-good based backtracking are:

1. Derive $\neg(x_{j_1} = w_{j_1} \wedge x_{j_2} = w_{j_2} \wedge ... \wedge x_{j_n} = w_{j_n})$ whenever the no-good j is implied by a single constraint, where $w_i \in D_i$ and $x_i \in X$

```
   function GetNextBox
4.1 |  if ! ScanFurtherForNewTuple then
    |  |  return 0;
    |  end
    |  repeat
    |  |  if ExpandableOnADirection then
4.2 |  |  |  ExpandOnFirstExpandableDirection
    |  |  end
    |  until ! SuccesfullyExpanded
4.3 |  StoreProgressInfo
    |  return 1
   end.
```

Algorithm 4: *Aggregation Procedure*

2. Backtracking appears when, for a given variable y, all values v_1 through v_n in its domain are ruled out by no-goods $\sum_i \rightarrow y \neq v_i$. Based on these no-goods, the $\neg(\sum_1 \wedge ... \wedge \sum_n)$ no-good may be generated. $\sum_j$ is the notation for a set of assignments $x_{j_1} = w_{j_1} \wedge x_{j_2} = w_{j_2} \wedge ... \wedge x_{j_n} = w_{j_n}$.

We have noticed that certain no-goods that might need to be reconsidered immediately may be erased with PDB and GPB, when we backtrack due to the elimination of all the values in the domain of a variable. We show that the space remains polynomial even if those additional no-goods are maintained.

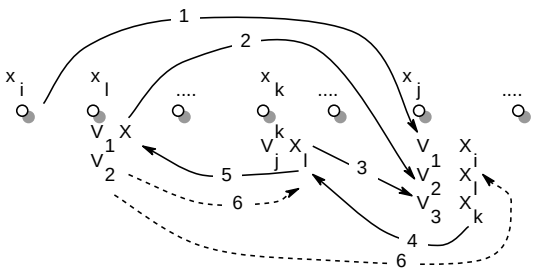
Let us consider the example in Figure 5 where the domain of variable x_j is exhausted and, based on the aforementioned rule, we generate a no-good that will have as conclusion $x_k \neq v_j^k$. The no-goods are marked with a cross that may be indexed by a variable in its elimination explanation. PDB and GPB perform first the elimination of all the no-goods that have as antecedent $x_k = v_j^k$. If it happens now that x_k gets also its whole domain eliminated, then based on the same rule we backtrack to a no-good with conclusion x_l. The modification of the value of x_l may eliminate some no-goods for values of x_j, besides the no-good for $x_k = v_j^k$.

We may decide to reinstantiate x_k to v_j^k. When we return from this backtracking we may be able to instantiate x_j and continue the search without rechecking the no-goods already existing for x_k. Therefore, the elimination of no-goods performed here by PDB and GPB was not appropriate. We have noticed, instead, that this behavior can be allowed by the structure of POB.

Definition 4 *An acceptable next assignment for a set of no-goods Γ and a set of variables X is any assignment that supports all those antecedents of the no-goods in Γ that do not depend on the variables in X. All the assignments of the variables that are not in X, or that are in X and are changed, do not figure in any no-good conclusion.*

The algorithm combining the described technique with the flexible ordering rules of GPB is called GOB and has the following two steps:

1. While no empty no-good is created and if one no-good γ is found, call =simp(γ,$\emptyset$)



x_i	x_l	x_k	x_j	new γ	GPB Γ	GOB Γ
v^i	v_1	v_j^k	v_1	$x_i \rightarrow x_j \neq v_1$ (1)	1	1
v^i	v_1	v_j^k	v_2	$x_l \rightarrow x_j \neq v_2$ (2)	1,2	1,2
v^i	v_1	v_j^k	v_3	$x_k \rightarrow x_j \neq v_3$ (3)	1,2,3	1,2,3
v^i	v_1	v_j^k	v_3	$x_i, x_l \rightarrow x_k \neq v_j^k$ (4)	1,2,4	1,2,3,4
v^i	v_2	v_j^k	v_3	$x_i \rightarrow x_l \neq v_1$ (5)	1,4,5	1,3,4,5
v^i	v_2	v_j^k	v_2			

Figure 5: *Sequence in GOB: the dashed arcs show no-goods removal while continuous lines show no-goods generation. The arcs are number in the order of the operation that they represent. In the first four columns of the table are the current assignments. The fifth column shows the no-goods added at each step while the sixth and seventh columns compare the list of no-goods maintained by GPB and GOB.*

2. If no empty no-good was inferred then return the current instantiation.

The procedure simp(γ,St) consists of the next steps:

1. If γ is empty then stop reporting no solution.

2. Use the translation rules to represent γ as an implication $\sum \rightarrow z \neq v_z$.

3. Add the no-good (represented as an implication) to the current set Γ of no-goods.

4. If the live domain of z is empty, it must be possible to apply the second inference rule for deriving new no-goods to obtain a no-good γ_1 involving variables constrained to precede z. Derive such a no-good and recursively backtrack from it by applying simp(γ_1,$St \cup \{z\}$).

5. Else, if no new no-good was generated in step 4., i.e., if the no-good added in step 3 did not eliminate all live values of z, then for each variable w, which must follow z, and for each x that must be followed by z under the current order constraints, add the the safety condition $x < w$ and delete all safety conditions of the form $y < w$

6. The assignment ρ is now consistent with any no-goods added in step 4. This implies that the live domain for z must now be nonempty. Set ρ to $\rho[z := v]$ where v is in the current live domain of z and remove all no-goods not relevant to the new assignment. Set any acceptable next assignment for Γ and St, and if variables in St are changed, remove all no-goods that are not relevant to the new assignment.

The rule required to translate a no-good to an implication will choose the conclusion of the implication as one of the variables in the no-good that is the last in at least a total order that complies with the current no-goods and with the current safety order.

The GOB algorithm has the qualities of GPB, while presenting the property we have described before.

Theorem 4 *The algorithm GOB is complete, terminates, is additive on disjoint subproblems for the first solution and visits at most as many states as GPB does.*

Proof. The GOB algorithm behaves like GPB with the only difference that the no-goods are erased only after making sure that some variables in their antecedents have changed. GPB guarantees a polynomial space by ensuring that all the no-goods are relevant and there exist at most one no-good per value. This means that we can have at most N^2K no-goods where N is the number of variables and K is the maximum domain size. For the current instantiation there exists at most NK no-goods (one for each value), and each no-good is at most of size N. In the case of GOB, since the additional no-goods we maintain remain relevant, the same reasoning holds. The existence of additional no-goods cannot increase the remaining space of the search at any step, therefore the maximum number of steps for GPB remains valid for GOB, and it terminates. The additivity is due to the same impossibility of generating no-goods between variables in disjunct problems as in GDB. For the comparison with GPB we notice that if we use with GOB the same heuristics for choosing no-goods, their conclusions and the reinstantiations, the only difference remains that some no-goods are not rediscovered in GOB for the case where variables are reinstantiated with the old values after backtracking. This reduces the number of expanded states. □

A difference to GPB is that using this approach, the algorithm cannot be requested to find a next assignment consistent with Γ at the end of each call to *simp* in the same way as GPB. When the set St is not empty, such instantiation cannot be found for the variables in it, which have an empty domain.

4.5 Dynamic Backtracking with CPR

We are designing algorithms for finding all solutions to CSPs. While the CPR [10] is shown to improve the performance for both finding all solutions and solving hard problems, we have decided to create a hybrid Box-GOB-CPR. In fact this algorithm is based on a instantiation of GOB. An implementation of CPR, based on the dual space representation, is proposed. CPR helps also in the compression of the generated solution.

By a no-good we mean one of the statements:

$$\neg(SetOfConstraints, SetOfValues, Variable)$$

and

$$\neg(SetOfConstraints, \neg Multibox, Constraint)$$

The algorithm is presented in Figure 6. The procedure SolutionNogood() generates a no-good for the last box, based on all the constraints. In order to have a cheaper management during the search, the matrix of the constraint representation is separated in sets of not yet scanned regions, no-goods and available boxes. The new no-goods are simplified at the step 4 by using the simp procedure of GOB.

Procedure COB(i)
If all constraints were instantiated (i=N),

- *report solution,*
- *SolutionNogood()*
- *return.*

Until the domain of a variable gets empty:

1. *Select a constraint c at position i or later if necessary. Change the ordering so that c is the ith position and delete conflicting order no-goods.*
2. - *Trim no-goods, unscanned regions and available regions. Add the eventual unscanned regions, by merging them to the existing ones.*
   - *If $\exists$ available regions, choose one*
     Else scan an unscanned region for a box.
3. *If a box was located,*
   - *expand it*
   - *create no-goods for the restricted variables and eliminate any no-goods pending on the expanded ones when compared with the previous assignment of c.*
   - *perform consistency maintaining over the future constraints*
   - *if a candidate box is obtained call $COB(i+1)$.*
4. *Else, if there is no other candidate multi-box in c,*
   - *derive a no-good based on all reducing constraints for all the variables in the current constraint and on all multi-boxes no-goods. Choose it's conclusion as the last in the current order over variables at positions $j < i$.*
   - *Simplify the new no-good and reorder the constraints by bringing the ones that have changed their instantiations after the current constraint, then update i.*

Figure 6: *C0B Algorithm*

5 CPR in ordered domains

The object of this section is twofold. We first motivate our choice of CPR by quantifying the gain it brings to the task of generating all solutions. This discussion is first presented for the original approach to CPR, based on the primal representation of CSPs. Once the usefulness of the approach has been stated, we go on to describe our dual-based implementation in ordered domains and compare it with the original approach.

5.1 On the Advantage of the Cross-Product Representation (CPR)

In [10] the gain brought by CPR is illustrated on several examples. We compute the number of checks spared by this approach in the general case of a discrete CSP with n-ary constraints. We show that a backtrack search process cannot perform worse with than without CPR.

Let b be a cross-product partial solution and v a not yet instantiated variable v. We use the following notations:

$NC(v, b)$ gives the number of constraints between the variable v and the already instantiated variables in b.

$ND(b)$, $Size_i(b)$ and $V(b)$ are respectively the number of instantiated variables, the number of values in the ith instantiated variable and the volume (number of partial solutions) of a cross-product partial solution b. By convention, the volume of an empty cross-product is 1.

$$V(b) = \prod_{i=1}^{ND(b)} Size_i(b) \tag{1}$$

$C_i(v, b)$ is the volume of the projection of b on the already instantiated variables in the ith constraint, C_i, among those counted by $NC(v, b)$.

Theorem 5 *Given a step of a search for all solutions of a discrete CSP, the CPR will turn the maximum number of checks for a current cross-product partial solution (cprps) at each instantiation (or Forward Check) of a new variable nv from:*

$$V(cprps) * NC(nv, cprps) \tag{2}$$

to:

$$\sum_{i=1}^{NC(nv,cprps)} C_i(nv, cprps) \tag{3}$$

for each active value (not yet eliminated by nogoods) in nv.

Proof. When looking for all solutions, by construction of the *cprps*, all the $V(cprps)$ partial solutions are checked and must be visited by a search without CPR. Any value of a new variable nv may have, therefore, to be tested for $NC(nv, cprps)$ constraints against each of the $V(cprps)$ partial solutions, leading to the Equation 2.

Using the CPR approach we only need to test each value of nv for each ith of the constraints counted by $NC(nv, cprps)$ against the $C_i(nv, cprps)$ values in the projected *cprps* cross-product and we obtain the Equation 3. □

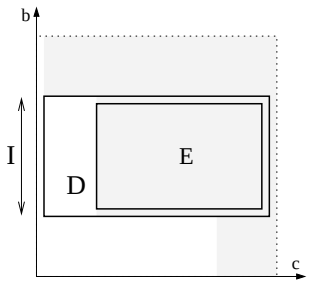


Figure 7: *If the variable b was instantiated by an interval I, the domain D where the merging E is performed for c is given by intersecting I with the range of b in the constraint between b and c*

Corollary 1 *For searching all solutions in a discrete CSP with constraints of arbitrary arities, a backtrack process cannot perform worse with than without CPR:*

$$\sum_{i=1}^{NC(nv,cprps)} C_i(nv, cprps) \leq V(cprps) * NC(nv, cprps)$$

Proof. We obtain the result from the definition of $C_i(nv, cprps)$,

$$C_i(nv, cprps) \leq V(cprps) \ \forall i, \tag{4}$$

by adding the inequalities 4 over all NC(nv,cprps).

□

5.2 CPR in Ordered Domains

We now compare the original approach of CPR with the dual-based implementation we propose. This implementation is referred to as DCPR in the rest of the paper. We first recall how the primal-based CPR works. Suppose that at a given step, $k-1$ variables $x_1, \ldots x_{k-1}$ have already been instantiated and that we want to assign aggregated values to a k^{th} new variable. The current instantiation for $x_1, \ldots x_{k-1}$ is structured as a partial cross-product, b_{k-1}. Each possible value v_i for x_k must be first tested against b_{k-1}, requiring the search for and the representation of cross-products of the type $b^i_{k-1} \times \{v_i\}$. The technique of CPR requires that these representations must be restructured by merging.

In our case, performing the instantiation in dual, directly provides the cross-products. These cross-products are in fact generated one by one and the merging step is implicit. The domains where the implicit merging is performed are determined by simple interval intersection, as shown in Figure 7. Moreover, the data management required by explicitly storing all partial cross-products is no more necessary. Performing DCPR in ordered domains also provides a concise representation for a cross-product, which takes the form of a set of intervals.

To summarize, the primal-based implementation of CPR reorders the values in the domains to perform the aggregations. As long as we deal with ordered domains, reordering is not needed with DCPR. CPR can benefit from the fact that the merging steps are more informed if the number of analyzed constraints between such steps is larger. The heuristics used by DCPR have however proved to be good enough to produce satisfactory results, as shown in the experiments.

DCPR offers the possibility of reordering the constraints during search allowing for more flexibility. A dual representation can incorporate any order of the values and any order of constraints while in the primal the order of the constraints is restricted by that of the variables. Note finally that DCPR is a full depth-first approach and hence is more likely to extend successfully to techniques with nogoods or backjumping. We now present the gain brought by DCPR, as done in the previous section for the primal-based CPR.

In the dual approach, we note with $C_i(b)$ the volume of the projection of b on the instantiated variables of the constraint i. The corresponding gain of DCPR is:

Theorem 6 *At each instantiation of a new constraint i, DCPR turns the maximum nr of checks for a current cross-product partial solution (cprps) from:*

$$V(cprps)/C_i(cprps) \tag{5}$$

to 1 for each active tuple (not yet eliminated by nogoods) in the constraint i.

Proof. At each constraint instantiation with DCPR, a tuple is tested once. In the standard tuple by tuple based approaches it would have been tested once for each combination of the values in the unrelated current labels. We mean the labels of all the instantiated variables not involved in the current constraint.

Here we have considered in $V(cprps)$ only the variables that were constrained by the past constraints.

This applies as well for the comparisons with the standard primal-based approaches if the order of testing the constraints is identical. □

From the definition of $C_i(b)$ we see that $V(cprps)/C_i(cprps)$ is always bigger than 1. For finding the first solution there is always a possible overhead since portions of the search space can uselessly be explored.

6 Heuristics

The aggregation procedure uses heuristics for assigning priorities to the directions of aggregation. The heuristic we propose is not designed to produce the multiboxes with the biggest volume. It rather tries to generate the multiboxes that promise the biggest further gain in the number of checks. The direction orders correspond to orders for the variables. Since the efficiency depends on the future constraints, we can use the result of the theorem 6 to estimate the gains of each direction of aggregation. We can therefore try to maximize the size of the labels for the variable that promises more. We estimate that the number of variable values that can be successfully aggregated along one variable v is proportional to the size of the current label of that variable $Size(v)$. Due to equation 5 we can estimate that for any future constraint C_i, for any of the variables v that are not implied in C_i, we will gain with a factor of $Size(v)$. This is the factor with which $V(cprps)$ of the equation 5 increases. If v is constrained by the constraint C_i, then there is no gain at the level of C_i. In our computation we say that for the variable v we have a *loss* of factor $Size(v)$.

One can therefore estimate that the *gain* of expanding the current constraint C_{crt} along direction v is proportional with

$$\sum_{i>crt, v \notin C_i} \frac{Size(v)V(cprps)}{C_i(cprps)}.$$

We have used a cheaper estimation of the *loss* of expanding the variable v as:

$$Size(v)(\sum_{i>crt, v\in C_i} 1). \quad (6)$$

7 Dual vs. Primal Approaches

Theorem 7 *For whatever run of a Chronological Backtracking (BT) [12] search on the primal representation with any strategy for variable ordering, there exists a strategy of constraint ordering and value ordering for which the BT in the dual representation performs identically.*

Proof. We show that for each run of a BT in the primal representation (PBT), there exists a run with identical checks of the BT for the dual representation (DBT). First we recall that in the primal approach to non-binary constraints[1], each constraint is checked only when all the referred variables are instantiated. We specify that in the dual approach, no check is being performed on the elements of the constraints that do not have the same values for the common variables with the already instantiated constraints.

We follow now a trace of the PBT and show how to build the corresponding DBT algorithm. If, when a variable is chosen to be instantiated in the PBT, in DBT we choose to instantiate the constraints in the order in which they are checked in PBT, the same checks will always be performed. Indeed, the PBT will try combinations in the first constraint until a first one is accepted. The same thing happens in DBT. Then PBT passes to the next constraint, and, if the value is not accepted, then it returns to the remaining values for previous one. The same happens in DBT, where after trying to instantiate the next constraint (same check as in PBT) we backtrack.

When variables in PBT can be instantiated with no check, then before passing its first check at the instantiation of a following variable, the order in respect to that variable is insured in DBT by establishing an order of the directions to scan the constraint tables. This way, the checks, as well as their order, are the same in the two algorithms. □

Corollary 2 *The Chronological Backtracking in Dual representation (DBT) is a generalization of the BT in the primal representation.*

Proof. The previous theorem has shown that DBT can emulate at no cost PBT. But no standard PBT can emulate a DBT that instantiates a variable before all the constraints between the already instantiated variables are checked. This is possible in DBT. □

8 Experiments

COB is currently under development. In this section, we present the experimental evaluation of Box-DCPR. Graph coloring problems were chosen for running the preliminary tests we report in this section. We consider the problem of coloring the n nodes of an indirected graph with C colors, so that no two nodes with the same color are linked by an edge. When all the possible colors for the nodes are ordered, the constraints of a graph coloring problem are matrices full of 1 entries, except for the elements of the main diagonal. This particularity makes them attractive for aggregation-based methods. The density of the order, as defined in the introduction is

affected by the number of colors in a graph coloring problem. The minimal density is 0 for 2-coloring problems, the maximal density is 2 since it cannot exceed the arity of the graph. A 3-coloring problem has a density of 0.66 while a 4-coloring has a density of 1.

The experiments where mainly run on 3-coloring problems, these problems having a relatively high density and sizes of domains reasonable enough to perform a significant number of tests.

The problems are generated in such a way that they have at least one solution and that they fall in the peak of difficulty, as described in [9]. More precisely, the graphs generated are *padded* coloring problems with *prespecified coloring* and an average of 4.9 arcs per node [9]. We have also taken into account the fact that for graph C-coloring problems, all assignments obtained by any permutation of the C colors in the given solution is also a solution.

It is therefore possible to fix the color of $C-1$ nodes about which we are sure that they will be colored differently, and all the solutions may be obtained by generating the permutations. Such $C-1$ nodes can be obtained by finding a clique of size $C-1$. A sequence of the resolution is to find the maximal clique of the graph. Let the size of the maximal clique be M. If $M > C$ then we can infer that there is no solution. If $M \leq C$ then we attribute M different colors to the nodes of the maximal clique and we search with DCPR for all solutions of the remaining problem. The other solutions can be found by generating all the arrangements of the M previously fixed colors with the values of the C colors. For each of the A_C^M arrangements of the M nodes, the arrangement of the other $C-M$ colors can be taken randomly, while the permutation will have been generated by the search itself. Changing the obtained permutations in the search solutions generates all the solutions of the initial problem.

The previously described approach still generates permutations that could have been inferred if $C-M > 1$. A solution were to integrate techniques for avoiding the permutations of the $C-M$ remaining colors into the search procedure. This is outside the goal of our tests. For the graphs with 3 colors, the case is not possible because any arc implies at least a clique of size 2. The clique is fixed by reducing the binary constraint to contain a single feasible pair.

The parameter we have used for our comparisons is the ratio between the running time of the algorithms on the same problems, on the same machines and with the same heuristics. We have run about 400,000 tests on problems with sizes between 20 and 100 variables. The results for each size of problem is averaged over about 10,000 instances. As expected, DCPR brings the better improvements for problems with large sizes. The gain brought in by the use of box-consistency maintenance is shown in the graph 8.

The tests shows that DCPR was generally better that the classical algorithm, with or without Box-Consistency. The ratio as a function of the size of the problem is shown in Figure 9. There was a small percentage $(0-5\%)$ of problems, generally very easy with a few number of variables, that performed better without aggregation. This is due to the fact that the aggregation procedure still has some overhead due to the ordering of directions and to multidimensional aggregations. The negative difference obtained in that case were less than seconds. However, some hard instances with the same parameters took days without aggregation and were solved in a few minutes with aggregations.

The behavior of Box-DCPR was also tested for finding the first solution. As described in Figure 10, Box-DCPR performs better than maintaining box-consistency without aggregation for hard problems. It was slightly over-performed only in some simple cases. The gain increases with the difficulty and with the size of the problem. We can therefore cautiously infer that Box-DCPR might be advisable for finding the first solution as well. This result was not expected since DCPR is likely to compute unnecessary solutions by aggregation.

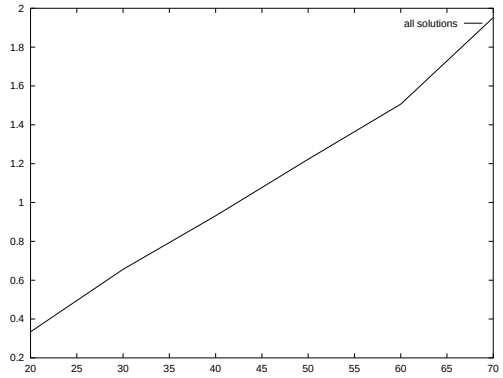


Figure 8: *Maintaining Box-Consistency: the average ratio between the time needed by DCPR with and without maintaining box-consistency vs. the number of variables. The scale is logarithmic. At 70 variable, Box-DCPR is in average about 90 times faster in finding all solutions.*

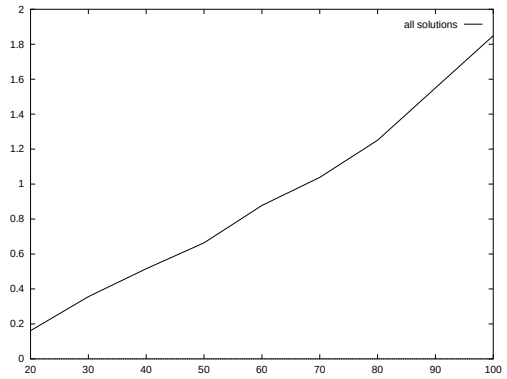


Figure 9: *Finding All the Solutions: the ratio between the time needed by the a classical backtracking with maintaining box-consistency and Box-DCPR vs. the number of variables. The scale is logarithmic. For example at 100 variable, Box-DCPR was in average about 80 times faster in finding all solutions.*

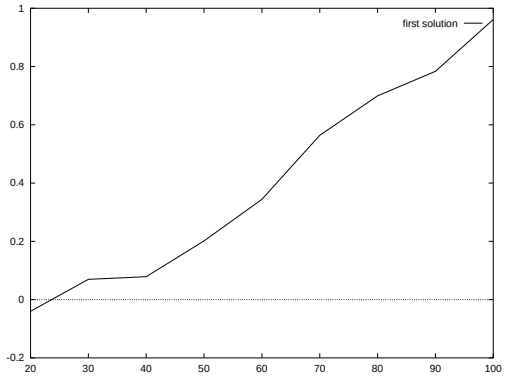


Figure 10: *Finding the First Solution: the average ratio between the time needed by a classical backtracking with maintaining box-consistency and Box-DCPR, vs. the number of variables. The scale is logarithmic scale. For example at 100 variable, DCPR was about 10 times faster in finding the first solution.*

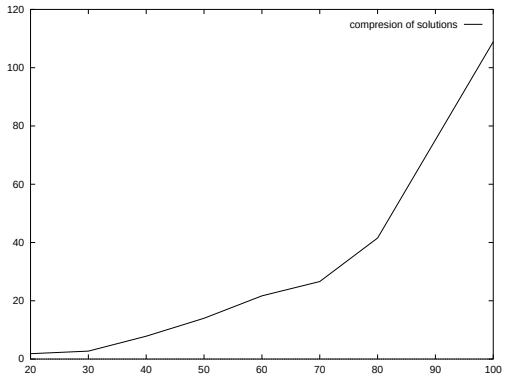


Figure 11: *Compression of the solution space: the ratio between the space needed to represent all the solutions by a search algorithm with maintaining box-consistency and DCPR vs. the number of variables.*

It was interesting to study the efficiency of the compression offered by DCPR for representing the solutions (Figure 11). As expected, the compression rate augments with the size of the problem.

We have also run some tests for 4 coloring problems with 30 variables. The average ratio was about 10 times better than the equivalent ratio for 3 colors.

9 Conclusion

This report proposed new search techniques for generating all solutions of ordered discrete CSPs. A central idea was to conjointly use intervals and cross-products for representing aggregation of values. This enabled the successful design of hybrid continuous-discrete search and local propagation techniques.

The explicit representation of constraints we used served as basis for designing intelligent splitting mechanisms, challenging the classical binary split. The solution space is compactly represented using CPR. We have shown that a dual based implementation of CPR offers many advantages such as allowing full depth approaches and merging on the fly as well as it spares unnecessary data management.

The encouraging results obtained by experimental evaluation of our look-ahead algorithm on graph coloring problems warrants the technique being further studied in other contexts.

In future work, we would like to study the usefulness of the approach in application with naturally ordered domains such as scheduling, numerical databases or hybrid continuous-discrete CSPs.

10 Acknowledgments

This work was performed at the Artificial Intelligence Laboratory of the Swiss Federal Institute of Technology in Lausanne and was sponsored by the Swiss National Science Foundation under project number 21-52462.97.

References

[1] F. Bacchus and P. van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *Proceedings of the 15th National Conference on Artificial Intelligence*, Madison, Wisconsin, July 98.

[2] A. Backer. The hazards of fancy backtracking. In *Proceedings of the 12th National Conference on AI*, pages 288–293. AAAI, 94.

[3] C. Bessiere. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, Jan 94.

[4] C. Bessière and J.-C. Régin. Mac and combined heuristics: Two reasons to forsake FC(and CBJ?) on hard problems. In *CP96*, pages 61–75. CP, 96.

[5] C. Bliek. Generalizing partial order and dynamic backtracking. In *AAAI-98 Proceedings*, pages 319–325, Madison,Wisconsin, July 98. AAAI.

[6] P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *ECAI'92*. ECAI, 92.

[7] M. Ginsberg and D. McAllester. Gsat and dynamic backtracking. In J.Doyle, editor, *Proceedings of the 4th IC on PKRR*, pages 226–237. KR, 94.

[8] P. Van Hentenryck. A gentle introduction to Numerica. *Artificial Intelligence*, 103(1-2):209–235, August 98.

[9] T. Hogg. Refining the phase transition in combinatorial search. *Artificial Intelligence*, 81(1-2):127–154, 96.

[10] P. D. Hubbe and E. C. Freuder. An efficient cross product representation of the constraint satisfaction problem search space. In *AAAI-92, Proceedings*, pages 421–427. AAAI, July 92.

[11] N. Jussien and O. Lhomme. Dynamic domain splitting for numeric CSP. In *European Conference on Artificial Intelligence*, pages 224–228, 98.

[12] G. Kondrak. A theoretical evaluation of selected backtracking algorithms. Master's thesis, Univ. of Alberta, 94.

[13] O. Lhomme and M. Rueher. Application des techniques CSP au raisonnement sur les intervalles. *Revue d'intelligence artificielle*, 11(3):283–311, 97. Dunod.

[14] D.A. McAllester. Partial order backtracking. *Research Note: 'ftp://ftp.ai.mit.edu/people/dam/dynamic.ps'*, 93.

[15] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings ECAI-94*, pages 125–129, 94.