Splitting Heuristics for Disjunctive Numerical Constraints

Thomas Douillard
LINA - Université de Nantes, France
Thomas.Douillard@univ-nantes.fr

Christophe Jermann
LINA - Université de Nantes, France
Christophe.Jermann@univ-nantes.fr

ABSTRACT

Numerical constraint solving techniques operate in a branch&prune fashion, using consistency enforcement techniques to prune the search space and splitting operations to explore it. Extensions address disjunctions of constraints as well, but usually in a restrictive case and not fitting well the branch&prune scheme. On the other hand, Ratschan has recently proposed a general framework for first-order formulas whose atoms are numerical constraints. It extends the notion of consistency to logical terms, but little is done with respect to the splitting operation. In this paper, we explore the potential of splitting heuristics that exploit the logical structure of disjunctive numerical constraint problems in order to simplify the problem along the search. First experiments on CNF formulas show that interesting solving time gains can be achieved by choosing the right splitting points.

Keywords

Numerical constraints; Disjunctions; Splitting heuristics

1. INTRODUCTION

Numerical constraint satisfaction problems (NCSP) are defined as a set of constraints on variables with values in domains that are subsets of $\mathbb{R}$. A solution to a NCSP is an assignment of values to the variables such that all the constraints are satisfied, i.e., the constraints are considered in conjunction. Though already expressive enough for a wide range of problems (e.g., in robotics, chemistry, computer-aided design or econometrics), the need for more expressive formalisms arise in several application fields: in design for instance, an artifact to be defined may be composed of alternative constrained elements, each element possibly being itself a composed constrained artifact; in molecular biology, genes activate or inhibit chemical reactions depending on their concentration; in formal program verification, the control structures enable alternative execution flows. Several formalisms have been introduced to address these applications (e.g. [8, 12]). In most cases they amount to introduce the possibility of expressing alternatives, guarded constraints, activity constraints, and other constructs that could well be expressed using logical connectors. It is then possible to express these problems as quantifier-free first-order logical formulas whose atoms are numerical constraints, which we call *disjunctive numerical constraint satisfaction problems* (DNCSPs) in the following.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil
Copyright 2008 ACM 978-1-59593-753-7/08/0003 ...$5.00.

A lot of work has been done on dealing with disjunctions of constraints, mostly in the discrete case [6, 7] but also in the continuous case [13, 4, 10]. However, in the latter case, most of the approaches consider a specific subclass on DNCSPs (e.g., [13] cannot handle non-linear equations, [4] considers only convex constraints) or follow a specific algorithmic scheme that cannot easily include the successful constraint programming (CP) techniques developped for NCSPs ([10] reasons on the logical structure following a SAT approach). On the other hand Ratchan [11] has proposed an extension of the CP framework to quantified first-order logical formulas whose atoms are numerical constraints. Neglecting the aspects of this framework that deal with quantifiers, it is possible to use it for DNCSPs. Though very general, this framework does not make much use of the logical structure of DNCSPs, concentrating its effort on handling quantifiers. However, the logical structure of DNCSPs let us expect a great potential in intelligent search strategies. Indeed, the logical structure of DNCSPs can be simplified along the search provided the right sub-search-spaces are isolated.

In this paper, we investigate this track of research and analyze the practical interest of intelligent splitting heuristics for DNCSPs. The rest of the paper is organized as follows: Section 2 recalls the necessary background; Section 3 introduces the concept of formula simplification; Section 4 explores the potential of intelligent splitting heuristics for DNCSPs and Section 5 presents the results for a selection of splitting heuristics on CNF formulas, a subclass of DNCSPs. Section 6 provides concluding remarks and future directions.

2. BACKGROUND

We denote F a quantifier-free first-order logical formula whose atoms are picked in a set C of numerical constraints (equations and inequalities) on a set X of variables, each variable x_i being associated with a domain $d_i \subseteq \mathbb{R}$. A solution to F is an assignment $\{x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n\}$ $(v_i \in d_i)$ such that the formula is satisfied by the assignment, i.e., the logical value of each atom (each constraint is either $true$ or $false$ on the assignment) makes the formula $true$. Such a formula can be expressed using solely conjunctions and disjunctions: any logical connector can be expressed using

only conjunction, disjunction and negation; negations can be distributed downto the atoms; the negation of an atom (a numerical constraint) can be expressed as another formula (e.g.,$\neg(a = b) \rightarrow (a < b) \vee (a > b)$). This is why we call it a *disjunctive numerical constraint problem.*

Interval solving techniques for NCSPs

In case the formula F is just the conjunction of the constraints in C, it is called a *numerical constraint satisfaction problem* (NCSP). The constraint programming approach to NCSPs uses interval arithmetic [9] in order to control the computation errors: each variable domain is represented as an interval, the search space being thus defined as an n-dimensional box. It yields correct and complete solving methods that follow a branch&prune scheme: the initial box B is pruned using *narrowing operators* based on *numerical local consistencies*, then split into several subboxes (usually two) each of which is recursively treated in the same way. See [3] for instance for more details.

Interval solving techniques for DNCSPs

Ratschan has proposed a general framework for solving quantified formulas on numerical constraints [11]. It defines extensions of classical numerical consistencies and their associated narrowing operators for each kind of logical term: conjunction, disjunction and quantification. Here we are interested only in the first two:

Given a numerical constraint consistency $\mathcal{C}$, a formula F is $\mathcal{C}$-consistent on a box B (denoted $\mathcal{C}(F, B)$) iff
– if $F = c$ (an atom) then $\mathcal{C}(F, B) = \mathcal{C}(c, B)$;
– if $F = \bigwedge F_i$ then $\mathcal{C}(F, B) = \bigwedge \mathcal{C}(F_i, B)$;
– if $F = \bigvee F_i$ then $\mathcal{C}(F, B) = \bigwedge \mathcal{C}(F_i, B_i)$ and $B = \mathcal{H}(\bigcup B_i)$ where $\mathcal{H}(S)$ is the smallest box enclosing a subset $S \in \mathbb{R}^n$ (called the *hull* of S).

The extension to atoms and conjunctions follows the standard definitions for NCSPs. The extension to disjunctions follows the principle of constructive disjunction [6, 7]: a box is consistent with a disjunction if it is the hull of the boxes which are consistent with its alternatives.

EXAMPLE 1. *Consider the disjunction of* 3 *numerical constraints* $c_1 : (x + 1)^2 + (y - 1)^2 = 1$, $c_2 : (x - 2)^2 + y^2 = 1$ *and* $c_3 : (x - 3)^2 + (y - 1)^2 = 1$ *on variables* x *and* y *with values in* $\mathbb{R}$ *(see Figure 1). The consistent domains for each individual constraint are:*

- $c1 \rightarrow x \in [-2, 0],\ y \in [0, 2]$
- $c2 \rightarrow x \in [1, 3],\ y \in [-1, 1]$
- $c3 \rightarrow x \in [2, 4],\ y \in [0, 2]$

The domain consistent with $c_1 \vee c_2 \vee c_3$ *for* x *(resp.* y*) is the hull of the union of the domains individually consistent with* c_1, c_2 *and* c_3 *for* x *(resp.* y*), i.e.:*

- $D_x = \mathcal{H}([-2, 0] \cup [1, 3] \cup [2, 4]) = [-2, 4]$
- $D_y = \mathcal{H}([0, 2] \cup [-1, 1] \cup [0, 2]) = [-1, 2]$

The obtained domain for x *contains a gap (see Figure 1), namely the open interval* $(0, 1)$ *which is inconsistent with all the alternatives of the disjunction.*

A weakness of the approach is that the logical structure of the formula is not exploited during the search. However the potential seems important since the pruning phase can infer interesting numerical properties that can impact the logical truth value of the formula: unsatisfiable atomic constraints, incompatible alternatives in disjunctions and even inconsistent inner-spaces (aka *gaps*) as illustrated in Example 1.

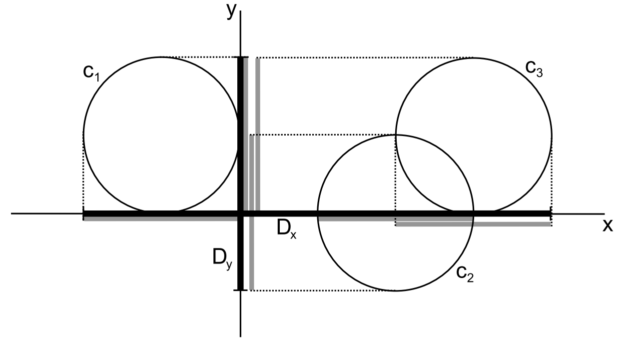

Figure 1: Consistency of a disjunction

Ratschan proposes an implementation optimization that discards the alternatives of disjunctions that have been proved inconsistent by pruning. However the information available after pruning could be used to simplify the formula further and to select interesting splitting points, e.g. isolating the different alternatives of a disjunction. This is what we present in the next sections.

3. SIMPLIFICATION OF DNCSPS

In this section, we formalize the concept of formula simplification which concretizes the connection between the numerical and logical levels of a DNCSP. Whenever the domain of a variable is modified, either by pruning or splitting, it is possible to simplify the DNCSP by computing the truth value of the atoms of the formula and propagating these truth values using four simple logical rules: $F \vee false \rightarrow F$, $F \vee true \rightarrow true$, $F \wedge false \rightarrow false$ and $F \wedge true \rightarrow F$.

An atom in a DNCSP is a numerical constraint, i.e., an equation or inequality. It is possible to determine that it is inconsistent (truth value = false) by projecting the constraint onto the domain of each of the variables it constrains; if one of the projections is empty, then the constraint is inconsistent. Similarly, it is possible to project its negation to determine if it is *totally* consistent (truth value = true), a technique that was introduced in order to implement inner consistencies [1]. If a constraint cannot be proved neither inconsistent nor totally consistent, then its truth value is undetermined.

The latter test is not very practical since inner consistency is rarely used, but the former one is exactly the operation performed by the narrowing operators employed in the pruning phase of the branch&prune algorithm. Hence, whenever a narrowing operator empties a domain during pruning, the associated constraint is determined inconsistent and the truth value propagation can be triggered, simplifying the formula accordingly.

EXAMPLE 2. *Consider again the formula presented in Example 1. In the box* $B = \{[1.5, 2.5], [0.5, 1.5]\}$ *this formula can be simplified into* $c_2 \vee c_3$*; indeed, as illustrated in Figure 2,* c_1 *is consistent in* $B_1 = \{[-2, 0], [0, 2]\}$ *but* $B_1 \cap B = \emptyset$*, hence* c_1 *can be removed, while* c_2 *and* c_3 *are respectively consistent in* $B_2 = \{[1, 3], [-1, 1]\}$ *and in* $B_3 = \{[2, 4], [0, 2]\}$*, boxes that both intersect* B*; hence* c_2 *and* c_3 *must be kept.*

Simplification seems a great tool, however it applies only when, by chance, an atom happen to be inconsistent. It is however possible to favorize the occurrences of such cases

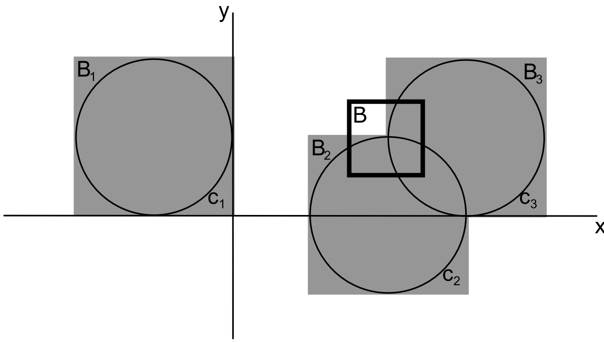

Figure 2: Simplification of a formula.

by adopting intelligent splitting heuristics that exploit the information gathered during the pruning phase.

4. SPLITTING HEURISTICS FOR DNCSPs

A splitting heuristic selects a variable domain to split and associated splitting points. It is the NCSP counterpart of the variable and value selection heuristics used in discrete CSPs. Given a splitting heuristic, the corresponding splitting operator produces a set of subboxes by replacing in the current box the selected domain by its subdomains between two consecutive spliting points. The traditional heuristic is roundrobin+bisection: domains are considered one after the other cyclically and they are split into two halves. More sophisticated heuristics have been proposed but they usually do not yield much gains except for specific classes of problems [5, 14, 2]. However, the potential gain in defining intelligent heuristics for DNCSPs appears great: a lot of information (e.g., gaps and incompatible alternatives) collected during the pruning operation is not used while it could induce interesting simplifications.

For this purpose, we introduce the concept of *interesting points* that represent potential splitting points and are defined as the bounds of the boxes consistent with each individual alternative in each disjunction of a formula. Indeed, these bounds define where each individual alternative is consistent (only at the right of lower bounds and at the left of upper bounds), i.e., where the gaps are (between consistent subdomains) and which alternatives are incompatible (non intersecting subdomains).

What makes a point "interesting" is that the corresponding disjunction is necessarily simplified in one of the subboxes resulting from splitting at this point: at least one alternative (the one whose lower (resp. upper) bound is the considered interesting point) is inconsistent in the left (resp. right) subdomain produced after splitting. Hence splitting at interesting points always induces simplification.

EXAMPLE 3. *After the pruning phase depicted in Figure 1 for the disjunction $F = c_1 \vee c_2 \vee c_3$ presented in Example 1, the interesting points are $\{0, 1, 2, 3\}$ for x and $\{0, 1\}$ for y. Suppose that we choose to split the domain of y. Using standard bisection the splitting point is 0.5 and the resulting subproblems cannot be simplified in the induced subboxes. On the contrary, splitting the same domain along its interesting points yield much more simplified subproblems:*
– $F \rightarrow c_2$ in the subbox $\{[-2, 4], [-1, 0]\}$),
– $F \rightarrow c_1 \vee c_2 \vee c_3$ in the subbox $\{[-2, 4], [0, 1]\}$),
– and $F \rightarrow c_1 \vee c_3$ in the subbox $\{[-2, 4], [1, 2]\}$).

We propose to define heuristics that select the actual splitting points among the interesting ones. In principle it could return several splitting points, yielding several subproblems and resulting in non-binary search trees. However, the number of interesting points depends on the number of disjunctions in the formula which is not bounded by its number of variables nor atomic constraints; hence it could be arbitrarily large, yielding memory consumption problems. In order to reduce the number of actual splitting points, it is possible to consider only the interesting points of a subformula (e.g., a single disjunction). Selecting the subformula, the variable domain to split and the actual splitting points among the considered interesting points is thus the key to obtain an efficient heuristic.

Numerous criteria can be used to help make this decision. They can be based, for example, on the induced simplification (local, i.e., only for the considered subformula, or global), the sizes and equilibrium of the induced subboxes, or the gaps they allow to elliminate. Instead of roaming among a huge number of criteria and theoretical considerations on them, we propose in the next section an experimental study of a handful of quite natural strategies.

5. EXPERIMENTAL ANALYSIS

We compare 4 intelligent heuristics and 2 witnesses. All use a first criterion to select a variable and a disjunction whose interesting points may become splitting points, and a second criterion to extract from the considered interesting points the actual splitting points. In the end, if the set of actual splitting points is empty they switch back to the standard splitting method: roundrobin+bisection.

Selected heuristics

- **largest gap** (LG): the selected pair (x, d) maximizes the volume of a single gap between the alternatives of d in the domain of x; the actual splitting points are the bounds of this gap. This heuristics produces binary search trees.

- **all gaps** (AG): the selected pair (x, d) maximizes the cumulated volume of all the gaps between the alternatives of d in the domain of x; the actual splitting points are the bounds of these gaps. This heuristics produces n-ary search trees.

- **all interesting points, all gaps** ($AIPAG$): the selected pair (x, d) maximizes the cumulated volume of all the gaps between the alternatives of d in the domain of x; the actual splitting points are *all* the interesting points of d relatively to x; hence, not only does this heuristic removes gaps but it also simplifies d as much as possible. It produces n-ary search trees.

- **all interesting points, roundrobin** ($AIPRR$): the variable x is selected in a roundrobin fashion and the disjunction d is a random disjunction involving x; the actual splitting points are all the interesting points of d relatively to x; thus, this heuristic performs a fair exploration of the search space (roundrobin), exploiting gaps if available and simplifying the formula as much as possible. It produces n-ary search trees.

Our two witnesses are bisection ($2S$) and *multisection* (KS). Both select the variable x in a round-robin fashion

and no disjunction; *2S* then splits the domain of x into two halves while *KS* splits it into k equilibrated parts, where k is the average number of alternatives per disjunction in the formula, i.e., an approximation of the number of possible interesting points. *KS* is here to ensure that the observed gains of intelligent strategies are not solely due to splitting into several small subboxes instead of only 2. Since they are not linked to any disjunction, neither *2S* nor *KS* perform simplification when splitting. Note that simplifying the formula with these heuristics would require testing the consistency of the atoms (contrarily to splitting along interesting points which already capture this information), a possibly costly operation which will whatsoever be performed in the next pruning phase.

Benchmark

In order to perform experiments on a broad scale, we have implemented a DNCSP generator that produces well-constrained problems composed of polynomial equations combined in a CNF formula with parameterizable numbers of solutions, variables, clauses and constraints per clauses. It generates 3 classes of problems: *poly* problems are composed of arbitrary polynomials; *uniform* problems are also composed of arbitrary polynomials but in each clause they all cover the same set of variables, a structure that augments the chances of gaps; *spheres* problems are composed of n-dimensional spheres (polynomials of degree 2), very structured problems with high chances of gaps and simplifications. In each class, we consider various generator parameterizations so that our benchmark offers a representative set of 14 problems and that the best heuristic solves most of these problems within 5 minutes, the time-out we have set.

Results

We have tested the 6 heuristics on a Pentium IV 2.8 GHz with 1Gb of memory (STU[1] = 33.64 seconds) with a tool implemented from scratch in C++. In order to obtain reliable results, we run each heuristic on 25 instances of each problem.

gen	2S	KS	LG	AG	RR-AIP	AG-AIP
poly1	2	12	1	1	0	0
poly2	8	**25**	9	9	5	9
poly3	3	**23**	2	2	2	3
poly4	0	**25**	1	1	5	3
poly5	0	**24**	0	0	4	4
uniform1	1	**15**	3	3	0	0
uniform2	**13**	**21**	12	12	8	0
uniform3	4	9	3	3	3	0
uniform4	0	0	0	0	2	0
uniform5	0	1	0	0	1	0
spheres1	0	2	0	0	**21**	1
spheres2	6	**25**	7	7	**25**	0
spheres3	10	**25**	10	10	**25**	6
spheres4	0	6	0	0	11	0
sum	*47*	*213*	*48*	*48*	*112*	*26*
avg	*13%*	*61%*	*14%*	*14%*	*32%*	*7%*

Table 1: Time-outs on 25 runs

Table 1 presents the number of time-outs per problems per heuristic. In bold in this table the cases with more time-outs than effective runs. The corresponding data are not presented in Table 2 since it would be unfair to compare average solving times on very different numbers of runs. The two gray lines at the end of this table respectively present the total number of time-outs per heuristic (line *sum*) and the average time-out ratio per heuristic (line *avg*). *KS* performs very poorly on more than half our benchmark. *AIPAG* seems to be the best heuristic except perhaps for *poly* problems. *AIPRR* performs poorly on *spheres* and *poly* problems. The other heuristics perform similarly.

	2S		KS		LG		AG		RR-AIP		AG-AIP	
gen	t	r	t	r	t	r	t	r	t	r	t	r
poly1	14	*1,36*	59	*5,69*	20	*1,92*	20	*1,96*	10	*1,00*	23	*2,26*
poly2	128	*1,28*	-	-	100	*1,00*	100	*1,00*	104	*1,05*	141	*1,42*
poly3	64	*1,00*	-	-	80	*1,25*	79	*1,25*	63	*1,00*	104	*1,63*
poly4	67	*1,11*	-	-	63	*1,04*	63	*1,04*	60	*1,00*	87	*1,45*
poly5	72	*1,21*	-	-	84	*1,41*	83	*1,41*	65	*1,10*	59	*1,00*
uniform1	65	*3,29*	-	-	58	*2,96*	59	*2,97*	40	*2,03*	20	*1,00*
uniform2	-	-	-	-	152	*3,11*	152	*3,11*	123	*2,51*	49	*1,00*
uniform3	111	*3,38*	129	*3,91*	117	*3,55*	116	*3,51*	64	*1,94*	33	*1,00*
uniform4	30	*3,01*	55	*5,44*	25	*2,47*	25	*2,47*	19	*1,94*	10	*1,00*
uniform5	55	*3,72*	53	*3,54*	55	*3,67*	54	*3,62*	31	*2,08*	15	*1,00*
spheres1	30	*1,00*	67	*2,23*	37	*1,22*	37	*1,22*	-	-	43	*1,41*
spheres2	141	*2,39*	-	-	149	*2,52*	150	*2,55*	-	-	59	*1,00*
spheres3	161	*1,29*	-	-	141	*1,13*	142	*1,13*	-	-	125	*1,00*
spheres4	26	*1,07*	38	*1,59*	24	*1,00*	28	*1,15*	26	*1,09*	40	*1,63*
max		*3,72*		*5,69*		*3,67*		*3,62*		*2,51*		*2,26*
avg		*1,93*		*3,73*		*2,02*		*2,03*		*1,52*		*1,27*
best		*1*		*0*		*2*		*0*		*3*		*8*

Table 2: Solving times in seconds

Table 2 presents the average solving time (white columns) and the ratio to the best solving time (gray columns) for the effective runs. The gray lines present respectively the maximum ratio per heuristic (line *max*), the average ratio per heuristic (line *avg*) and the number of times a heuristic has been the best (line *best*). Again, *AIPAG* appears to be the best and most stable heuristic: it obtains the highest number of *best*, has an average ratio close to one (indicating that it usually performs in times comparable to the best heuristic) and its worst ratio (2.26) is attained for one *poly* problem only whereas for all others it is less than 2. *AIPRR* obtains very good results too, but we have to keep in mind that it reaches the time-out very often when compared to *AIPAG* (see Table 1); this emphasizes its unstability, even in a single problem class like *spheres*. *KS* confirms the bad results already shown in Table 1: it is never the best heuristic and regularly performs in more than twice the best time. The other heuristics obtain similar results.

Analysis of the results

From these results and additional information we gathered about the number of splitting operations, the number of interesting points exploited, the number of simplifications due to splitting, the volume of the eliminated gaps (information we cannot reproduce here for space reasons) we can extract the following trends:

- Sophisticated splitting heuristics are not expensive and can yield interesting gains. In fact, though the splitting times for intelligent heuristics (*AG*, *AIPAG*, *AIPRR*) are sometimes 30 times bigger than that of bisection, it remains in any case negligible in comparison to the total solving time (usually less than 2%). This cost is balanced by the gains these heuristics allow not only in times but also tree depths and number of iterations of the branch&prune algorithm.

[1] http://www.mat.univie.ac.at/ neum/glopt/coconut/Benchmark/Benchmark.html

- Exploiting only the gaps (LG, AG) does not bring any gain compared to bisection. In fact, these heuristics apply rarely: they fall back to standard bisection in 99% cases in average for *poly* problems and 95% in average for the two other classes of problems. A careful look at the logs indicates that very few gaps appear during the solving once the initial ones have been exploited.

- The gains observed using the interesting points ($AIPAG$, $AIPRR$) does not come solely from the fact these heuristics yield numerous small subboxes. Indeed, KS is also a heuristic that splits boxes into numerous subboxes but it obtains very poor results. This indicates that choosing the splitting points among the interesting points is crucial for obtaining good performances when performing n-ary splits.

- Using all the interesting points ($AIPRR$, $AIPAG$) is better than only the gaps (LG, AG). Indeed $AIPRR$ and $AIPAG$ apply more than 70% of the cases in average, i.e., they have a real impact on the solving, while the other two often fall back to bisection. A thorough study of the logs show that the simplifications obtained using all the interesting points at each split help make appear gaps in later iterations of the algorithm. This is why $AIPAG$ applies more than 90% of the cases on $uniform$ and $spheres$ problems.

6. CONCLUSION

We have formalized the connection between the logical and numerical aspects of DNCSPs and proposed a simplification principle that fits well in the branch&prune algorithm.

We have also investigated the potential of splitting heuristics that exploit the logical structure of DNCSPs in order to achieve greater simplification. This potential is confirmed by first experiments on structured and unstructured problems of polynomial equations in CNF form. The results are promising and open broad directions of research.

Our experiments show that interesting points can bring good gains even if they do not bound a gap, but it is still difficult to distinguish the really interesting ones from the others. This is however crucial for larger problems: preliminary experiments on large problems (tens of variables and constraints per clauses) show that the number of interesting points explodes and then using all of them as done by $AIPRR$ and $AIPAG$ is not practicable anymore. Future works will address this limitation.

Preliminary experiments on problems issued from design and control applications also show promising results. For instance, we solved a conceptual design problem presented in [15] modeled as a DNSCP with 19 variables, 6 disjunctions (from 2 to 20 alternatives each), and 182 atomic constraints (14 non-linear). $2S$ solves the problem in 5.84 seconds and is the worse heuristic for this problem; $AIPAG$ solves the problem in 2.37 seconds and is the best heuristic for this problem. This problem has a very specific, yet common in design applications, structure. Indeed, some of its disjunctions represent catalog constraints, i.e., discrete choices among sets of prefabricated components with different characteristics. The consistant domains of these disjunctions are composed of isolated points with large holes in between. This explains why $AIPAG$ is so efficient on this problem. We expect similar results on other design and control applications which have such specific structures.

7. REFERENCES

[1] F. Benhamou and F. Goualard. Universally quantified interval constraints. In *6th International Conference on Principles and Practice of Constraint Programming (CP'2000)*, in LNCS 1894, pages 67–82, 2000.

[2] G. Chabert. *Techniques d'intervalles pour la résolution de systèmes d'équations*. PhD thesis, Université de Nice-Sophia Antipolis, 2007.

[3] D. M. F. Benhamou and P. Van Hentenryck. CLP(Intervals) revisited. In *International Logic Programming Symposium*, pages 124–138, 1994.

[4] I. Grossmann. Review of nonlinear mixed-integer and disjunctive programming techniques.

[5] C. M. H. Batnini and M. Rueher. Mind the gaps: A new splitting strategy for consistency techniques. In *11th International Conference on Principles and Practice of Constraint Programming*, 2005.

[6] P. V. Hentenryck. *Constraint satisfaction in logic programming*. MIT Press, 1989.

[7] O. Lhomme. An efficient filtering algorithm for disjunction of constraints. In *9th International Conference on Principles and Practice of Constraint Programming*, pages 904–908, 2003.

[8] S. Mittal and B. Falkenhaimer. Dynamic constraint satisfaction problems. In *8th National Conference on Artificial Intelligence*, pages 25–32, 1990.

[9] R. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[10] A. O. R. Nieuwenhuis and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[11] S. Ratschan. Efficient solving of quantified inequality constraints over the real numbers. *ACM Transactions on Computational Logic*, 7(4):723–748, 2006.

[12] D. Sabin and E. Freuder. Configuration as composite constraint satisfaction. In *Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161, 1996.

[13] M. Salido and F. Barber. A polynomial algorithm for continuous non-binary disjunctive csps: extended dlrs. *Knowledge-Based Systems*, 16(5-6):277–285, 2003.

[14] J. S.-H. X.-H. Vu, M. Silaghi and B. Faltings. Branch-and-prune search strategies for numerical constraint solving. Technical Report LIA-REPORT-2006-007, Swiss Federal Institute of Technology (EPFL), 2006.

[15] A. Anglada. *Introduction de mécanisme de flexibilité dans les contraintes de domaines continues*. PhD thesis, Lip6, Université de Paris 6, October 2005.