Support inference for generic filtering

Frederic Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais

CRIL (Centre de Recherche en Informatique de Lens)
CNRS FRE 2499
rue de l'université, SP 16
62307 Lens cedex, France
{boussemart,hemery,lecoutre,sais}@cril.univ-artois.fr

Abstract. In this paper, we show that a static analysis of constraints can deliver interesting structural knowledge. Such knowledge, viewed as some general constraint properties, is very useful in achieving support inference, thereby reducing the cost of consistency checking. Indeed, exploiting during search some properties established in a preprocessing step can lead to substantial improvements of backtracking algorithms that maintain a form of local consistency such as arc consistency.

1 Introduction

One challenge of the Constraint Programming (CP) community is to develop efficient general approaches to solve CSP instances. As constraint propagation is recognized as an important component in the design of efficient search procedures, many filtering techniques have been introduced by the CP community. In this paper, we are interested in exploiting new general constraint properties in order to improve generic filtering, thereby reducing the gap between generic and specific approaches.

Arc consistency is a general property of constraint networks which can be established stand-alone or maintained during search. When arc consistency is maintained during search, an initialization phase is generally achieved by a preprocessing step, and a propagation phase is repeated at each step of the search process. The idea behind this paper is to propose new improvements in order to reduce the cost of the propagation phase. In other words, the idea is to achieve substantial computations during the preprocessing step in order to avoid useless consistency checking during search.

More precisely, we present a static analysis of the structure of each constraint in order to extract interesting properties, namely, conflict sets, covering sets and substitutability. Then, we show how it is possible to exploit such knowledge in a search algorithm that maintains a domain filtering consistency such as arc consistency.

2 Formal background

An instance of the Constraint Satisfaction Problem (CSP) is defined by a constraint network. A constraint network is a pair $(\mathscr{X},\mathscr{C})$ where $\mathscr{X}$ is a finite set of variables such that each variable $X \in \mathscr{X}$ has an associated domain $dom(X)$ denoting the set of values allowed for X, and $\mathscr{C}$ is a finite set of constraints such that each constraint $C \in \mathscr{C}$ has an associated relation $rel(C)$ denoting the set of tuples allowed for the variables $vars(C)$ involved in the constraint C.

Let C be a k-ary constraint such that $vars(C) = \{X_{i_1}, \ldots, X_{i_k}\}$, $dom(C)$ represents the domain of C, i.e. the Cartesian product $\prod_{j=1}^{k} dom(X_{i_j})$. A k-tuple t is a support of C iff $t \in rel(C)$ and a conflict of C iff $t \in dom(C) - rel(C)$. The set of supports of C restricted to $X = a$ is denoted $supports(C)_{X=a}$ and the set of conflicts of C restricted to $X = a$ is denoted $conflicts(C)_{X=a}$

Now, we introduce some properties (given in the binary case) that can be exploited by any backtracking algorithm that alternates variable instantiation and constraint propagation enforcing a domain filtering consistency [5] such as arc-consistency, restricted path consistency, singleton arc consistency, etc. The properties that we have identified permit inferences that reduce the cost of looking for constraint supports. As a result, the number of constraint checks performed by propagation can be significantly decreased.

A first mechanism to infer supports involves (the cardinality of) conflict sets.

Proposition 1. *Let $C \in \mathscr{C}$ s.t. $vars(C) = \{X, Y\}$ and $a \in dom(X)$. At any step of the search, if $|dom(Y)| > \lambda$ then $supports(C)_{X=a} \neq \emptyset$ where λ is the size of the initial set of conflicts of C restricted to $X = a$.*

A second mechanism to infer supports is based on covering sets.

Definition 1. *Let $C \in \mathscr{C}$ s.t. $vars(C) = \{X, Y\}$ and $S \subseteq dom(Y)$. S is a covering set from Y for C iff $\forall a \in dom(X), \exists b \in S$ s.t. (a, b) is a support of C.*

Proposition 2. *Let $C \in \mathscr{C}$ s.t. $vars(C) = \{X, Y\}$ and S a covering set from Y for C. At any step of the search, if any element of S still remains in $dom(Y)$ then $\forall a \in dom(X)$ $supports(C)_{X=a} \neq \emptyset$.*

The last mechanism is called substitutability that has been introduced by [6] and defined with respect to all constraints of a CSP. In this paper, we are interested in a restricted form of substitutability that is defined independently for each constraint of a CSP. Restricting such a property to a given constraint might lead to a great number of substitutable values.

Definition 2. *Let $C \in \mathscr{C}$, $X \in vars(C)$ and $\{a, b\} \subseteq dom(X)$. a is substitutable for b wrt X in C, denoted $a \succeq_{C,X} b$, iff $supports(C)_{X=a} \supseteq supports(C)_{X=b}$.*

Proposition 3. *Let $C \in \mathscr{C}$, $X \in vars(C)$ and $a \in dom(X)$.*

- *$supports(C)_{X=a} \neq \emptyset \Rightarrow \forall b \in dom(X)$ s.t. $b \succeq_{C,X} a$, $supports(C)_{X=b} \neq \emptyset$.*
- *$supports(C)_{X=a} = \emptyset \Rightarrow \forall b \in dom(X)$ s.t. $a \succeq_{C,X} b$, $supports(C)_{X=b} = \emptyset$.*

3 Support inference in arc consistency algorithms

It is possible to exploit the properties that have been identified in the previous section when using an algorithm establishing arc consistency. More precisely, we have chosen to illustrate our approach with respect to AC3 [9]. Indeed, AC3 is quite simple to present, and all extensions introduced here can be directly integrated in other coarse-grained arc consistency algorithms such as AC2001/3.1, AC3.2 and AC3.3 (and also, for some of them, adapted to fine-grained ones).

We have developed the three following versions:

- $AC3^{R0}$ corresponds to AC3.
- $AC3^{R1}$ is the variant of AC3 that exploits conflict sets and covering sets.
- $AC3^{R2}$ is the variant of AC3 that exploits conflict sets, covering sets and substitutability.

The two variants of AC3 that we have introduced above are not very interesting when one just establishes arc consistency. Indeed, the initialization phase is heavy, and then penalizing if no search is performed. Clearly, the variants we propose must be used with a MAC (or GAC) algorithm. Initializing the new data structures before or after preprocessing and exploiting them all along search is a good compromise between the cost of the initialization and the benefit that can be obtained.

4 Experiments

To prove the practical interest of the properties introduced in this paper, we have implemented the different algorithms described in the previous section and conducted an experimentation with respect to significant classes of random and real-world instances. Performances have been measured in terms of the number of constraint checks (#ccks) and the cpu time in seconds (cpu). Also, on some problems involving 100 instances, performance criteria are given on average.

We have used a MAC algorithm which maintains arc consistency by using AC3 [9], AC2001/3.1 [3, 11] or AC3.2 [8]. Also, we have used the variable ordering heuristic $dom/wdeg$ [4] as it has been shown to be a quite efficient generic heuristic.

First, we have experimented some realistic scheduling problems. We have selected 4 representative instances from the set of 60 job shop instances proposed by [10]. One can observe on Table 1 the significant gain obtained wrt AC3 when exploiting unidirectionality (AC3.1/AC2001) and (partial) bidirectionality (AC3.2), conflict and covering sets (R1) and substitutability (R2). As illustrated by the behavior of $AC3.2^{R2}$, on this example the more exploited constraint knowledge is, the more efficient the search algorithm is.

Finally, to study the behaviour of the different variants wrt random problems, we have considered five classes $<n, d, m, t>$ of 100 random instances located at the phase transition where n denotes the number of variables, d the uniform domain size, m the number of binary constraints and t the constraint tightness

instances		$MAC3$		$MAC3.1$		$MAC3.2$	
		cpu	#ccks	cpu	#ccks	cpu	#ccks
$<RG = 0.1, BK = 1>$	$R0$	251.64	1,449,387K	192.22	611,045K	98.37	242,679K
(sat)	$R1$	123.26	600,751K	91.38	266,557K	80.81	215,979K
	$R2$	100.55	375,148K	80.59	173,698K	75.69	152,896K
$<RG = 0.1, BK = 2>$	$R0$	374.49	2,281,435K	169.28	522,253K	116.64	291,574K
(sat)	$R1$	262.59	1,448,883K	145.67	427,742K	113.24	278,557K
	$R2$	189.27	778,953K	147.63	257,601K	116.98	172,229K
$<RG = 0.0, BK = 1>$	$R0$	497.57	2,883,037K	350.90	1,158,383K	175.21	425,669K
(sat)	$R1$	228.72	1,042,013K	176.96	493,222K	146.39	383,073K
	$R2$	179.69	695,099K	147.10	335,314K	131.83	277,671K
$<RG = 0.0, BK = 2>$	$R0$	418.46	2,454,923K	222.26	727,814K	154.72	429,273K
(sat)	$R1$	298.89	1,563,852K	194.98	588,483K	152.37	398,453K
	$R2$	226.73	880,159K	188.48	341,490K	159.91	241,383K

Table 1. Job shop instances

(number of unallowed tuples). These classes have been selected such that the tightness ranges from 5% to 88%. It is interesting to note in Table 2 that the smaller the tightness is, the more efficient variants $R1$ and $R2$ are.

instances		$MAC3$		$MAC3.1$		$MAC3.2$	
		cpu	#ccks	cpu	#ccks	cpu	#ccks
$< 30, 10, 1330, 5 >$	$R0$	39.52	58,253K	46.47	32,047K	42.98	16,611K
(32/100 sat)	$R1$	26.40	2,723K	28.93	2,636K	27.99	1,738K
	$R2$	27.42	2,409K	29.52	2,331K	28.54	1,512K
$< 30, 14, 725, 20 >$	$R0$	93.20	150,315K	105.16	80,536K	92.25	42,699K
(49/100 sat)	$R1$	57.17	24,380K	71.56	20,916K	68.14	15,228K
	$R2$	66.10	22,357K	79.20	18,951K	77.69	13,597K
$< 50, 10, 494, 20 >$	$R0$	208.39	348,812K	240.57	196,167K	198.85	97,260K
(70/100 sat)	$R1$	157.22	109,300K	197.82	85,754K	180.89	55,579K
	$R2$	186.94	97,946K	222.95	75,997K	209.97	48,575K
$< 40, 20, 200, 175 >$	$R0$	14.49	25,037K	12.32	12,827K	9.10	6,848K
(51/100 sat)	$R1$	12.28	16,918K	11.63	9,977K	9.60	6,243K
	$R2$	13.73	16,359K	12.88	9,705K	11.12	6,123K
$< 150, 50, 230, 2200 >$	$R0$	26.78	60,057K	14.70	25,687K	9.95	13,089K
(31/100 sat)	$R1$	24.34	51,818K	14.62	24,348K	9.79	13,569K
	$R2$	24.11	50,183K	14.84	23,726K	10.39	13,389K

Table 2. random instances

5 Related work

An arc consistency algorithm schema, AC-Inference, has been described [2] that allows exploiting generic and specific properties of constraints so as to save useless constraint checks. AC7 is a derivation of AC-Inference that simply utilizes bidirectionality. We have shown, in this paper, that (a local form of) substitutability is another generic constraint property that allows support inference. Hence, it should be possible to build from AC-Inference a general-purpose fine-grained arc consistency algorithm exploiting both bidirectionaly and substitutability.

Interchangeability and its extended forms has been introduced by [6]. Simply removing interchangeable values at the preprocessing stage has been experimentally shown productive under certain conditions [1]. And using interchangeability independently for each constraint in order to save constraint checks has been proposed by [7]. Our exploitation of substitutability can then be seen as a refinement of the piece of work of [7] related to constraint propagation.

6 Conclusion

In this paper, we have introduced new general properties that allow support inference for any backtracking algorithm maintaining a domain filtering consistency. Exploiting such properties can be seen as a tentative to reduce the gap between generic and specific approaches.

We have shown that coarse-grained arc-consistency algorithms can significantly benefit from these properties since the variants that we have presented:

– are easy to implement (new data structures are not dynamically updated),
– have a limited space requirement ($O(md)$ versus $O(md^2)$ for AC-Inference),
– are competitive.

Acknowledgements

This paper has been supported by the CNRS, the “programme COCOA de la Région Nord/Pas-de-Calais” and by the “IUT de Lens”.

References

1. B.W. Benson and E.C. Freuder. Interchangeability preprocessing can improve forward checking search. In *Proceedings of ECAI'92*, pages 28–30, 1992.
2. C. Bessière, E.C. Freuder, and J. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148, 1999.
3. C. Bessière and J. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01*, pages 309–315, 2001.
4. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04 (to appear)*, 2004.
5. R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.
6. E.C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of AAAI'91*, pages 227–233, 1991.
7. A. Haselbock. Exploiting interchangeabilities in constraint satisfaction problems. In *Proceedings of IJCAI'93*, pages 282–287, 1993.
8. C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings of CP'03*, pages 480–494, 2003.
9. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:118–126, 1977.
10. N. Sadeh and M.S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86:1–41, 1996.
11. Y. Zhang and R.H.C. Yap. Making AC3 an optimal algorithm. In *Proceedings of IJCAI'01*, pages 316–321, Seattle WA, 2001.