Extending Generalized Arc Consistency

Anastasia Paparrizou and Kostas Stergiou

Department of Informatics and Telecommunications Engineering,
University of Western Macedonia, Greece

Abstract. Generalized arc consistency (GAC) is the most widely used local consistency in constraint programming. Several GAC algorithms for specific constraints, as well as generic algorithms that can be used on any constraint, have been proposed in the literature. Stronger local consistencies than GAC have also been studied but algorithms for such consistencies are generally considered too expensive. In this paper we propose an extension to the standard GAC algorithm `GAC2001/3.1` that achieves a stronger local consistency than GAC by considering intersections of constraints. Importantly, the worst-case time complexity of the proposed algorithm, called `GAC+`, is higher than that of `GAC2001/3.1` only by a factor e, where e is the number of constraints in the problem. Experimental results demonstrate that in many cases `GAC+` can reduce the size of the search tree compared to GAC, resulting in improved cpu times. Also, in cases where there is no gain in search tree size, there is only a negligible overhead in cpu time.

1 Introduction

Constraint Programming (CP) has become one of AI's success stories in recent years and is nowdays an established paradigm for modelling and solving hard combinatorial problems from areas such as planning and scheduling, timetabling, resource allocation, bioinformatics, etc. At the core of CP's success is the wide range of efficient contraint propagation algorithms offered by modern CP solvers for several types of non-binary constraints.

Constraint propagation algorithms typically try to prune values from the domains of variables by enforcing a local consistency property on the constraints of the problem. The most widely used local consistency is generalized arc consistency (GAC), also known as domain consistency. Several specialized GAC algorithms for specific (*global*) constraints have been proposed in the literature. In the absense of specialized algorithms, generic GAC algorithms that can be used on any constraint are applied. Examples of such algorithms are `GAC3` [9], `GAC4` [10], and `GAC2001/3.1` [1].

Stronger local consistencies than GAC have also been studied extensively. Although the application of such methods can result in significant gains in terms of search tree size, they are rarely used in CP solvers. This is because algorithms for strong consistencies are generally considered too expensive, meaning that potential gains in search tree size are very often outweighted by the cpu time overheads.

In this paper we propose an extension to the well-known generic GAC algorithm `GAC2001/3.1` that achieves a stronger local consistency than GAC through the intelligent exploitation of simple data structure used by `GAC2001/3.1`. Importantly, the

I. Maglogiannis, V. Plagianakos, and I. Vlahavas (Eds.): SETN 2012, LNAI 7297, pp. 174–181, 2012.

worst-case time complexity of GAC+, is higher than that of GAC2001/3.1 only by a factor e, where e is the number of constraints in the problem. Experimental results from benchmark problems demonstrate that in many cases GAC+ can reduce the size of the search tree compared to GAC, resulting in improved cpu times. Also, in cases where there is no gain in search tree size, there is only a negligible overhead in cpu time.

2 Background

A *Constraint Satisfaction Problem* (CSP) is defined as a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where: $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of n variables, $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is a set of finite domains, one for each variable, with maximum cardinality d, and $\mathcal{C} = \{c_1, \ldots, c_e\}$ is a set of e constraints with maximum arity k. Each constraint c is a pair $(vars(c), rel(c))$, where $vars(c) = \{x_1, \ldots, x_m\}$ is an ordered subset of $\mathcal{X}$, and the relation $rel(c)$ is a subset of the *Cartesian* product $D(x_1) \times \ldots \times D(x_m)$ that specifies the allowed combinations of values for the variables in $var(c)$. Constraints are represented either extensionally by explicitly specifying their relation or intensionally through a predicate or function.

Each tuple $\tau \in rel(c)$ is an ordered list of values $(a_1, \ldots, a_k)$ such that $a_j \in D(x_j)$,$j = 1, \ldots, k$. Given a constraint c, a variable $x_i \in var(c)$, and a tuple $\tau \in rel(c)$, we denote by $\tau[x_i]$ the projection of τ on x_i. A tuple $\tau \in rel(c)$ is *valid* iff none of the values in the tuple has been removed from the domain of the corresponding variable.

Given two constraints c_i and c_j, if $var(c_i) \cap var(c_j) \neq \emptyset$ then we say that the constraints *intersect*. We denote by f_{max} the maximum number of variables that are common to any two constraints that share more than one variable.

A standard way of solving CSPs is by interleaving depth-first search and constraint propagation. The former is typically guided by branching heuristics, while the latter involves repeatedly enforcing some *local consistency* property on the constraints of the problem so that infeasible values are located and pruned.

The most commonly used local consistency is *generalized arc consistency* (GAC) or *domain consistency*, simply referred to as *arc consistency* in the case of binary constraints.

Definition 1. A value $a_i \in D(x_i)$ is GAC iff for every constraint c s.t. $x_i \in vars(c)$ there exists a valid tuple $\tau \in rel(c)$ that includes the assignment of a_i to x_i. In this case τ is called a *support* of a_i. A variable is GAC iff all its values are GAC. A problem is GAC iff there is no empty domain in $\mathcal{D}$ and all the variables in $\mathcal{X}$ are GAC.

When applied, GAC and its weaker variants such as Bounds Consistency (BC), focus on one constraint at a time. In contrast, higher-order local consistencies exploit the fact that very often constraints have two or more variables in common, to achieve stronger pruning than GAC. One of the most promising such consistencies is *max Restricted PairWise Consistency* (maxRPWC) [2].

Definition 2. A value $a_i \in D(x_i)$ is maxRPWC iff $\forall c_j \in C$, where $x_i \in var(c_j)$, a has a support $\tau \in rel(c_j)$ s.t. $\forall c_l \in C$ $(c_l \neq c_j)$, s.t. $var(c_j) \cap var(c_l) \neq \emptyset, \exists \tau' \in$

$rel(c_l)$, s.t. $\tau[var(c_j) \cap var(c_l)] = \tau'[var(c_j) \cap var(c_l)]$ and τ' is valid. In this case we say that τ' is a PW-support of τ. A variable is maxRPWC iff all values in its domain are maxRPWC. A problem is maxRPWC iff there is no empty domain in $\mathcal{D}$ and all variables are maxRPWC.

Although the application of maxRPWC, and other strong local consistencies, can result in stronger pruning than GAC, it is not widely used. This is due to the high cost of algorithms for maxRPWC. For example, the standard maxRPWC1 algorithm has $O(e^2k^2d^p)$ worst-case time complexity, where p is the maximum number of variables involved in two constraints that share at least two variables [2]. In contrast, `GAC2001/3.1` has $O(ek^2d^k)$ worst-case complexity which is significantly lower considering that $p = 2k - 2$ in the worst case.

Following [4] we call a local consistency A stronger than B iff in any problem in which A holds then B holds, and strictly stronger iff it is stronger and there is at least one problem in which B holds but A does not.

3 GAC+

We now present `GAC+`, an algorithm that extends `GAC2001/3.1` to achieve a local consistency stronger than GAC. Specifically, when `GAC+` is applied it deletes all values that are not GAC and in addition it can delete some extra values that are GAC but are not maxRPWC. To achieve this it utilizes the $LastGAC$ data structure of `GAC2001/3.1`. To recall the use of this data structure, for each constraint c and each value $a_i \in D(x_i)$, where $x_i \in var(c)$, $LastGAC_{c,x_i,a_i}$ gives (i.e. points to) the most recently discovered support of a_i in c.

Algorithm 1. Algorithm GAC+

```
1:  if PREPROCESSING then L=L ∪ {x_i}, ∀x_i ∈ V;
2:  else L={ currently assigned variable };
3:  while L ≠ ∅ do
4:     L=L−{x_i};
5:     for each c_k ∈ C s.t. x_i ∈ var(c_k) do
6:        for each x_j ∈ V s.t. x_j ∈ var(c_k) AND x_j ≠ x_i do
7:           if revise_GAC+(c_k, x_j) > 0 then
8:              if DWO(x_j) then return FAILURE;
9:              L=L ∪ {x_j};
10: return SUCCESS;
```

Algorithm `GAC+` utilizes a list L where variables that have their domains pruned are inserted. Once a variable x_i is extracted from L, each constraint c_k that involves x_i is examined (line 5 in Algorithm 1) and all the variables that appear in c_k, except x_i, are revised. This is done by calling Function *reviseGAC+*.

This function takes a constraint c_i and a variable x_j, s.t. $x_j \in var(c_i)$, and for each value $a_j \in D(x_j)$ first checks if a_j has a support in c_i. In case $LastGAC_{c_i,x_j,a_j}$ is valid then this tuple is a support for a_j. If $LastGAC_{c_i,x_j,a_j}$ is not valid anymore, a

new support is seeked. This is done by iterating through the tuples of c_i in lexicographical order starting from the one immediatelly after $LastGAC_{c_i,x_j,a_j}$ (line 5 in Function 2). In case a tuple τ that is valid and consistent is located, then a support for a_j has been established and $LastGAC_{c_i,x_j,a_j}$ is set to τ. Up to this point GAC+ operates just like a typical GAC algorithm. However, once a support τ is located, GAC+ performs an additional operation which can sometimes determine that τ has no PW-support in some intersecting constraint. Namely, the algorithm iterates over the constraints intersecting with c_i on more than one variable and for each such constraint c_k calls function *checkPWtuple*[1].

Function 2. reviseGAC+(c_i, x_j)

```
1:  removedValues = 0;
2:  for each a_j ∈ D(x_j) do
3:     SUPPORT_FOUND=FALSE;
4:     if ¬isValid(LastGAC_{c_i,x_j,a_j}) then
5:        for each τ of c_i > LastGAC_{c_i,x_j,a_j}, s.t. τ[x_j] = a_j do
6:           if isValid(τ) AND isConsistent(c_i, τ) then
7:              LastGAC_{c_i,x_j,a_j} = τ;
8:              PW_CONSISTENCY=TRUE;
9:              for each c_k ≠ c_i s.t. |var(c_k) ∩ var(c_i)| > 1 do
10:                if checkPWtuple(c_i, τ, c_k) then
11:                   PW_CONSISTENCY=FALSE; break;
12:             if PW_CONSISTENCY then
13:                SUPPORT_FOUND=TRUE; break;
14:       if ¬ SUPPORT_FOUND then
15:          remove a_j from D(x_j);
16:          removedValues = removedValues + 1;
17: return removedValues;
```

This function first locates Lex_Max, the lexicographically largest $LastGAC_{c_k,x_k,\tau[x_k]}$ for all variables x_k that belong to the intersection of c_i and c_k (lines 1-4). Then it checks if there can exist a tuple greater or equal to this one that has the same values for the variables of the intersection as τ. Crucially, this check is done in linear time as follows.

Assuming $Lex\_Max =< (x_1, a_1), ..., (x_m, a_m) >$ then this tuple is scanned from left to right. If the currently examined variable x_k belongs to $var(c_k) \cap var(c_i)$ and $a_k > \tau[x_k]$, where a_k is the value of x_k in Lex_Max, then we conclude that there can be no PW-support for τ in c_k (line 10). In the opposite case where $a_k < \tau[x_k]$ (line 10), we infer that a PW-support could be located for τ and thus we stop searching. If x_k does not belong to $var(c_k) \cap var(c_i)$ then if the value it takes in Lex_Max is the last value in its domain, we continue scanning (line 7). Otherwise, the scan is stopped because there may exist a tuple larger or equal to Lex_Max that is a PW-support of τ.

As implied by its description, *checkPWtuple* can verify the lack of PW-support mainly in cases where the variables in the intersection appear consecutively at the start of

[1] Constraints that intersect on exactly one variable are not considered because after making the problem GAC they cannot possibly contribute to any extra pruning [2].

constraint's c_k scope. Hence, this function performs a limited, and cheap, check for PW consistency. That is, it can sometimes determine that a verified support τ is not PW consistent (i.e. it has no PW-support on some constraint). In such a case, the search for a support for a_j is resumed in *reviseGAC+*. The following example illustrates the basic idea behind GAC+.

Function 3. checkPWtuple(c_i, τ, c_k)

```
1: Lex_Max=NULL;
2: for each x_k ∈ var(c_k) ∩ var(c_i) do
3:     if τ' = LastGAC_{c_k,x_k,τ[x_k]} > Lex_Max then
4:         Lex_Max=τ';
5: for each x_k ∈ var(c_k) do
6:     if x_k ∉ var(c_k) ∩ var(c_i) then
7:         if Lex_Max[x_k] is last value in D(x_k) then continue;
8:         else break;
9:     else
10:        if Lex_Max[x_k] < τ[x_k] then break;
11:        if Lex_Max[x_k] > τ[x_k] then return FALSE;
12: return TRUE;
```

Example 1. Consider two constraints c_1 and c_2 with $var(c_1) = \{x_1, x_2, x_3, x_4\}$ and $var(c_2) = \{x_3, x_4, x_5, x_6\}$. Assume that the support $\tau = \{0, 2, 2, 1\}$ has been located for value 0 of x_1, and that $LastGAC_{c_2,x_3,2}$ is tuple $\tau' = \{2, 2, 0, 1\}$. Since $\tau'[x_4]$ is greater than $\tau[x_4]$, it is clear that there is no valid and consistent tuple in c_2 that includes values 2 and 1 for x_3 and x_4 respectively. That is, no PW-support for τ exists in c_2 and hence value 0 of x_1 is not maxRPWC. If we assume that τ is the last support of $(x_1, 0)$ in c_1 then GAC+ will determine (simply by comparing τ to τ') that 0 should be deleted from $D(x_1)$. In contrast, a GAC algorithm cannot infer this since it does not consider constraint intersections at all.

The following proposition is a direct consequence of the limited check for PW consistency that GAC+ performs. The proof is straightforward if we consider that GAC+ is identical to GAC2001/3.1 plus the calls to Function *checkPWtuple*, which can only result in extra pruning, and is thus ommitted.

Proposition 1. GAC+ achieves a level of local consistency that is strictly stronger than GAC and strictly weaker than maxRPWC.

As mentioned, the ability of GAC+ to delete extra values compared to a GAC algorithm depends on the ordering of the variables in the scope of the constraints. For instance, if the scope of constraint c_2 in Example 1 is $var(c_2) = \{x_3, x_5, x_4, x_6\}$ with $LastGAC_{c_2,x_3,2}$ being $\tau' = \{2, 0, 2, 1\}$ then we cannot deduce that no PW-support for τ exists in c_2 unless 0 is the last value in $D(x_5)$. This is because a tuple that is lexicographically greater than τ', e.g. $\{2, 1, 1, 1\}$ may be a PW-support of τ. However, the ordering of the constraints' scope can be altered if necessary. For example, if a subset of the variables in a constraint appears in many intersections with other constraints then these variables can be moved to the front of the constraint's scope to facilitate pruning by GAC+. This can be done for all constraints in a preprocessing step.

Finally, we discuss the worst-case complexity of GAC+. Since GAC+ uses the same $LastGAC$ data structure as GAC2001/3.1, it has the same O(ekd) space complexity.

Proposition 2. The worst-case time complexity of GAC+ is $O(e^2k^2d^k)$.

Proof. GAC+ is identical to GAC2001/3.1 with the addition of lines 8-13 to *reviseGAC+*. In *reviseGAC+*, for each variable x_j and each of its d values, d^{k-1} tuples are first checked for GAC consistency with O(k) cost for each check. Then, for each tuple and each constraint c_k interecting c_i *checkPWtuple* is called.

Let us now consider the cost of *checkPWtuple*. Finding the lexicographically largest $LastGAC$ among the at most f_{max} variables in $var(c_k) \cap var(c_i)$ costs O(f_{max}), assuming that the lexicographic comparison of two tuples is implemented efficiently. The **for** loop of line 5 costs O(k) since in the worst case all values in the tuple must be examined. Hence, the cost of *checkPWtuple* is O($f_{max} + k$)=O(k).

Hence, *reviseGAC+* costs O($dd^{k-1}(k+ek)$) = O(ekd^k). This function can be called at most kd times for each constraint c_i and variable $x_j \in var(c_i)$. However, the cost of *reviseGAC+* for each x_j and each c_i is amortized over all the kd calls because of the use of $LastGAC$ (see [1] for details). Since there are at most e constraints and k variables per constraint, the worst-case time complexity of GAC+ is O($e^2k^2d^k$). □

4 Experiments

We ran experiments with benchmark non-binary problems taken from C. Lecoutre's repository and used in the CSP Solver Competitions[2]. We tried the following classes: *Golomb rulers*, *random problems*, *forced random problems*, *chessboard coloration*, *Schurr's lemma*, *modified Renault*, *positive table constraints* and *BDD*. The first five classes only include constraints of arity up to 4, while the other three include constraints of large arity (up to 18).

The algorithms were implemented within a CP solver written in Java from scratch. Search used a binary branching scheme, the *dom/wdeg* heuristic for variable ordering [3], and lexicographical value ordering. The searches for GAC on extensional constraints of large arity were performed using the efficient algorithm of [6]. The ordering of variables in the constraint scopes was not altered to facilitate propagation for GAC+, although this is an interesting direction for future work.

In Table 1 we present indicative results from search algorithms that maintain a certain local consistency throughout search. We compare GAC+ to GAC (implemented using algorithm GAC2001/3.1). The results demonstrate that GAC+ improves uppon the performance of GAC2001/3.1 in the majority of instances.

Specifically, GAC+ is clearly better than GAC2001/3.1 on *Golomb rulers* instances as well as *random* and *forced random* problems. Often there are large margins between the performances of the two algorithms. For example on *rand-3-20-20-60-632-fcd-15* GAC+ is 3 times faster than GAC2001/3.1. These results are due to the stronger pruning achieved by GAC+ which results in significant reduction in the number of nodes.

[2] http://www.cril.univ-artois.fr/CPAI08/

Table 1. Search tree nodes and cpu times in secs from various representative problem instances

Instance	Node visits		CPU time	
	GAC2001/3.1	GAC+	GAC2001/3.1	GAC+
renault-mod-5	1,070	1,038	**326**	332
renault-mod-10	1,532	1,514	48	**47**
renault-mod-24	753	674	217	**206**
renault-mod-25	1,273	545	510	**365**
renault-mod-31	863	796	76	**69**
bdd-21-133-18-78-6	41,199	39,002	3,521	**2,777**
bdd-21-133-18-78-7	36,383	31,713	**4,312**	4,462
ruler-25-8-a4	2,697	2,316	96	**67**
ruler-34-9-a4	8,495	9,430	1,264	**934**
rand-3-20-20-60-632-fcd-4	223,155	113,814	275	**154**
rand-3-20-20-60-632-fcd-8	136,912	110,585	171	**145**
rand-3-20-20-60-632-fcd-15	85,940	25,858	109	**35**
rand-3-20-20-60-632-4	124,450	37,612	165	**51**
rand-3-20-20-60-632-7	114,375	112,592	**150**	155
rand-3-20-20-60-632-9	73,408	48,956	102	**67**
pt-8-20-5-18-800-4	37,466	37,416	1,301	**1,181**
pt-8-20-5-18-800-7	15,845	15,757	505	**464**
cc-8-8-2	13,278	13,762	**7.2**	7.8
cc-9-9-2	12,945	12,828	**12**	13
lemma-20-9	370,992	370,992	**101**	102
lemma-30-9	367,664	367,664	**249**	253

GAC+ does not achieve notable additional pruning on *positive table constraints*. Albeit, it is still faster than GAC2001/3.1. Results are somewhat mixed on the *modified Renault* and *BDD* classes. However, GAC+ is faster than GAC2001/3.1 in the majority of the instances.

GAC+ is not successful, in terms of pruning, on the *chessboard coloration* and *Schurr's lemma* classes. This is due to the structure of the instances in these classes. In *chessboard coloration* constraints have relatively small arity (4) and they are very loose (disjunctions of $\neq$ constraints). This minimizes the extra pruning that can be achieved by GAC+. Note that in some cases GAC+ results in more node visits than GAC2001/3.1, meaning that its few extra value deletions actually mislead the variable ordering heuristic. In *Schurr's lemma* problems there are only a few constraint intersections on more than one variable. As a result, our method cannot exploit the problems' structure for additional pruning. However, despite the lack of additional pruning in these two classes, the overheads of GAC+ do not slow down search notably compared to GAC2001/3.1.

4.1 Discussion

From the experimental results we can conclude that the performance of GAC+ depends largely on the structure of the particular problem class, i.e. on the topology of the constraint graph and the type of constraints. Our method is particularly successful on

problems where many intersections between constraints exist, and the constraints are relatively tight. On the other hand, GAC+ does not offer improvements on problems with few intersections or/and when constraints are loose. We believe that these results can be exploited to preselect the appropriate propagation technique by examining the structure of the given problem.

It is important to note that the idea on which GAC+ is based (i.e. the exploitation of the $LastGAC$ data structure) is not only applicable within a generic algorithm. Algorithms for certain specialized constraints can also benefit. Specifically, specialized GAC algorithms for *table constraints* (i.e. extensionally defined constraints) can be enhanced to achieve stronger pruning following the ideas presented here in a straightforward way. The GAC algorithms of [8], [7] and [5] already utilize a structure similar to $LastGAC$. Therefore, it is easy to extend them along the lines of GAC+. We intend to investigate this in the future.

5 Conclusion

We have presented GAC+, an extension to the standard GAC algorithm GAC2001/3.1 that achieves a stronger local consistency level than GAC. This is accomplished through the exploitation of a simple data structure already used by GAC2001/3.1. In contrast to existing methods for strong local consistencies, the worst-case time complexity of GAC+ is very close to that of GAC algorithms. This is reflected on the practical performance of the algorithm as it does not slow down search in a significant way even in cases where no additional pruning compared to GAC is achieved. On the other hand, there exist cases where the additional pruning of GAC+ results in important cpu time gains.

References

1. Bessière, C., Régin, J.C., Yap, R., Zhang, Y.: An Optimal Coarse-grained Arc Consistency Algorithm. Artificial Intelligence 165(2), 165–185 (2005)
2. Bessiere, C., Stergiou, K., Walsh, T.: Domain filtering consistencies for non-binary constraints. Artificial Intelligence 172(6-7), 800–822 (2008)
3. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: Proceedings of ECAI 2004, Valencia, Spain (2004)
4. Debruyne, R., Bessière, C.: Domain Filtering Consistencies. JAIR 14, 205–230 (2001)
5. Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Data structures for generalised arc consistency for extensional constraints. In: Proceedings of the Twenty Second Conference on Artificial Intelligence (2007)
6. Lecoutre, C.: Optimization of Simple Tabular Reduction for Table Constraints. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 128–143. Springer, Heidelberg (2008)
7. Lecoutre, C., Szymanek, R.: Generalized Arc Consistency for Positive Table Constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 284–298. Springer, Heidelberg (2006)
8. Lhomme, O., Régin, J.C.: A fast arc consistency algorithm for n-ary constraints. In: Proceedings of AAAI 2005 (2005)
9. Mackworth, A.K.: On reading sketch maps. In: Proceedings IJCAI 1977, pp. 598–606 (1977)
10. Mohr, R., Masini, G.: Good Old Discrete Relaxation. In: Proceedings of ECAI 1988, pp. 651–656 (1988)