Adaptive Parameterized Consistency

Amine Balafrej, Christian Bessiere, Remi Coletta, El Houssine Bouyakhf

Adaptive Parameterized Consistency*

Amine Balafrej[1,2], Christian Bessiere[1], Remi Coletta[1], El Houssine Bouyakhf[2]

[1]CNRS, University of Montpellier, France
[2]LIMIARF/FSR, University Mohammed V Agdal, Rabat, Morocco
{balafrej,bessiere,coletta}@lirmm.fr, bouyakhf@fsr.ac.ma

Abstract. State-of-the-art constraint solvers uniformly maintain the same level of local consistency (usually arc consistency) on all the instances. We propose *parameterized local consistency*, an original approach to adjust the level of consistency depending on the instance and on which part of the instance we propagate. We do not use as parameter one of the features of the instance, as done for instance in portfolios of solvers. We use as parameter the *stability* of values, which is a feature based on the state of the arc consistency algorithm during its execution. Parameterized local consistencies choose to enforce arc consistency or a higher level of local consistency on a value depending on whether the stability of the value is above or below a given threshold. We also propose a way to dynamically adapt the parameter, and thus the level of local consistency, during search. This approach allows us to get a good trade-off between the number of values pruned and the computational cost. We validate our approach on various problems from the CSP competition.

1 Introduction

Enforcing constraint propagation by applying local consistency during search is one of the strengths of constraint programming (CP). It allows the constraint solver to remove locally inconsistent values. This leads to a reduction of the search space. Arc consistency is the oldest and most well-known way of propagating constraints [2]. It has the nice feature that it does not modify the structure of the constraint network. It just prunes infeasible values. Arc consistency is the standard level of consistency maintained in constraint solvers. Several other local consistencies pruning only values and stronger than arc consistency have been proposed, such as max restricted path consistency or singleton arc consistency [7]. These local consistencies are seldom used in practice because of the high computational cost of maintaining them during search. However, on some problems, maintaining arc consistency is not a good choice because of the high number of ineffective revisions of constraints that penalize the CPU time. For instance, Stergiou observed that when solving the scen11 radio link frequency assignment problem (RLFAP) with an algorithm maintaining arc consistency, only 27 out of the 4103 constraints of the problem were identified as causing a domain wipe out and 1921 constraints did not prune any value [10].

* This work has been funded by the EU project ICON (FP7-284715).

Choosing the right level of local consistency for solving a problem requires finding the good trade-off between the ability of this local consistency to remove inconsistent values, and the cost of the algorithm that enforces it. Stergiou suggests to take advantage of the power of strong consistencies to reduce the search space while avoiding the high cost of maintaining them in the whole network. His method results in a heuristic approach based on the monitoring of propagation events to dynamically adapt the level of local consistency (arc consistency or max restricted path consistency) to individual constraints. This prunes more values than arc consistency and less than max restricted path consistency. The level of propagation obtained is not characterized by a local consistency property. Depending on the order of propagation we can converge on different closures. When dealing with global constraints, some work propose to weaken arc consistency instead of strengthening it. In [8], Katriel et al. proposed a randomized filtering scheme for AllDifferent and Global Cardinality Constraint. In [9], Sellmann introduced the concept of approximated consistency for optimization constraints and provided filtering algorithms for Knapsack Constraints based on bounds with guaranteed accuracy.

In this paper we define the notion of stability of values. This is an original notion not based on characteristics of the instance to solve but based on the state of the arc consistency algorithm during its propagation. Based on this notion, we propose *parameterized consistencies*, an original approach to adjust the level of consistency inside a given instance. The intuition is that if a value is hard to prove arc consistent (i.e., the value is not stable for arc consistency), this value will perhaps be pruned by a stronger local consistency. The parameter p specifies the threshold of stability of a value v below which we will enforce a higher consistency to v. A parameterized consistency p-LC is thus an intermediate level of consistency between arc consistency and another consistency LC, stronger than arc consistency. The strength of p-LC depends on the parameter p. This approach allows us to find a trade-off between the pruning power of the local consistency and the computational cost of the algorithm that achieves it. We apply p-LC to the case where LC is max restricted path consistency. We describe the algorithm p-maxRPC3 (based on maxRPC3 [1]) that achieves p-max restricted path consistency. Then, we propose ap-LC, an adaptive variant of p-LC which adapts dynamically and locally the level of local consistency during search. Finally, we experimentally assess the practical relevance of parameterized local consistency. We show that by making good choices for the parameter p we take advantage of both arc consistency light computational cost and LC effectiveness of pruning. In the best cases, a solver using p-LC explores the same number of nodes as LC with a number of constraint checks lower than LC, resulting in a CPU-time lower than both arc consistency-based or LC-based solvers.

2 Background

A *constraint network* is defined as a set of n variables $X = \{x_1, ..., x_n\}$, a set of ordered domains $D = \{D(x_1), ..., D(x_n)\}$, and a set of e constraints

$C = \{c_1, ..., c_e\}$. Each constraint c_k is defined by a pair $(var(c_k), sol(c_k))$, where $var(c_k)$ is an ordered subset of X, and $sol(c_k)$ is a set of combinations of values (tuples) satisfying c_k. In the following, we restrict ourselves to binary constraints because the local consistency (maxRPC) we use here to instantiate our approach is defined on the binary case only. However, the notions we introduce can be extended to non-binary constraints, by using maxRPWC for instance [4]. A binary constraint c between x_i and x_j will be denoted by c_{ij}, and $\Gamma(x_i)$ will denote the set of variables x_j involved in a constraint with x_i.

A value $v_j \in D(x_j)$ is called an *arc consistent support (AC support)* for $v_i \in D(x_i)$ on c_{ij} if $(v_i, v_j) \in sol(c_{ij})$. A value $v_i \in D(x_i)$ is *arc consistent (AC)* if and only if for all $x_j \in \Gamma(x_i)$ v_i has an AC support $v_j \in D(x_j)$ on c_{ij}. A domain $D(x_i)$ is arc consistent if it is non empty and all values in $D(x_i)$ are arc consistent. A network is arc consistent if all domains in D are arc consistent. If enforcing arc consistency on a network N leads to a domain wipe out, we say that N is arc inconsistent.

A tuple $(v_i, v_j) \in D(x_i) \times D(x_j)$ is *path consistent (PC)* if and only if for any third variable x_k there exists a value $v_k \in D(x_k)$ such that v_k is an AC support for both v_i and v_j. In such a case, v_k is called *witness* for the path consistency of (v_i, v_j).

A value $v_j \in D(x_j)$ is a *max restricted path consistent (maxRPC)* support for $v_i \in D(x_i)$ on c_{ij} if and only if it is an AC support and the tuple (v_i, v_j) is path consistent. A value $v_i \in D(x_i)$ is max restricted path consistent on a constraint c_{ij} if and only if $\exists v_j \in D(x_j)$ maxRPC support for v_i on c_{ij}. A value $v_i \in D(x_i)$ is max restricted path consistent iff for all $x_j \in \Gamma(x_i)$ v_i has a maxRPC support $v_j \in D(x_j)$ on c_{ij}. A domain $D(x_i)$ is maxRPC if it is non empty and all values in $D(x_i)$ are maxRPC. A network is maxRPC if all domains in D are maxRPC.

We say that a local consistency LC_1 is stronger than a local consistency LC_2 ($LC_2 \preceq LC_1$) if LC_2 holds on any constraint network on which LC_1 holds.

The problem of deciding whether a constraint network has solutions is called the *constraint satisfaction problem (CSP)*, and it is NP-complete. Solving a CSP is done by backtrack search that maintains some level of consistency between each branching step.

3 Parameterized Consistency

In this section we present an original approach to parameterize a level of consistency LC stronger than arc consistency so that it degenerates to arc consistency when the parameter equals 0, to LC when the parameters equals 1, and to levels in between when the parameter is between 0 and 1. The idea behind this is to be able to adjust the level of consistency to the instance to be solved, hoping that such an adapted level of consistency will prune significantly more values than arc consistency while being less time consuming than LC.

Parameterized consistency is based on the concept of stability of values. We first need to define the 'distance to end' of a value in a domain. This captures

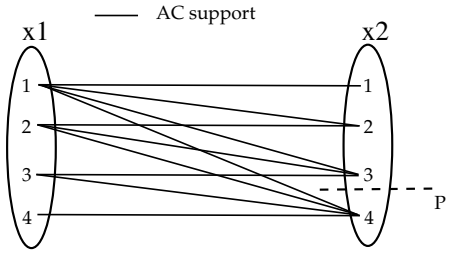

Fig. 1. Stability of supports on the example of the constraint $x_1 \leq x_2$ with the domains $D(x_1) = D(x_2) = \{1, 2, 3, 4\}$. $(x_1, 4)$ is not p-stable for AC.

how far a value is from the last in its domain. In the following, $rank(v, S)$ is the position of value v in the ordered set of values S.

Definition 1 (Distance to end of a value). *The* distance to end *of a value* $v_i \in D(x_i)$ *is the ratio*

$$\Delta(x_i, v_i) = (|D_o(x_i)| - rank(v_i, D_o(x_i)))/|D_o(x_i)|,$$

where $D_o(x_i)$ *is the initial domain of* x_i*.*

We see that the first value in $D_o(x_i)$ has distance $(|D_o(x_i)|-1)/|D_o(x_i)|$ and the last one has distance 0. Thus, $\forall v_i \in D(x_i), 0 \leq \Delta(x_i, v_i) < 1$.

We can now give the definition of what we call the parameterized stability of a value for arc consistency. The idea is to define stability for values based on the distance to the end of their AC supports. For instance, consider the constraint $x_1 \leq x_2$ with the domains $D(x_1) = D(x_2) = \{1, 2, 3, 4\}$ (see Figure 1). $\Delta(x_2, 1) = (4 - 1)/4 = 0.75$, $\Delta(x_2, 2) = 0.5$, $\Delta(x_2, 3) = 0.25$ and $\Delta(x_2, 4) = 0$. If $p = 0.2$, the value $(x_1, 4)$ is not p-stable for AC, because the first and only AC support of $(x_1, 4)$ in the ordering used to look for supports, that is $(x_2, 4)$, has a distance to end smaller than the threshold p. Proving that the pair $(4, 4)$ is inconsistent (by a stronger consistency) could lead to the pruning of $(x_1, 4)$. In other words, applying a stronger consistency on $(x_1, 4)$ has more chances to lead to its removal than applying it on for instance $(x_1, 1)$, which had no difficulty to find its first AC support (distance to en of $(x_2, 1)$ is 0.75).

Definition 2 (p-stability for AC). *A value* $v_i \in D(x_i)$ *is p-stable for AC on* c_{ij} *iff* v_i *has an AC support* $v_j \in D(x_j)$ *on* c_{ij} *such that* $\Delta(x_j, v_j) \geq p$*. A value* $v_i \in D(x_i)$ *is p-stable for AC iff* $\forall x_j \in \Gamma(x_i)$*,* v_i *is p-stable for AC on* c_{ij}*.*

We are now ready to give the first definition of parameterized local consistency. This first definition can be applied to any local consistency LC for which the consistency of a value on a constraint is well defined. This is the case for instance for all triangle-based consistencies [6,2].

Definition 3 (Constraint-based p-LC). *Let LC be a local consistency stronger than AC for which the LC consistency of a value on a constraint is defined. A value $v_i \in D(x_i)$ is constraint-based p-LC on c_{ij} iff it is p-stable for AC on c_{ij}, or it is LC on c_{ij}. A value $v_i \in D(x_i)$ is constraint-based p-LC iff $\forall c_{ij}$, v_i is constraint-based p-LC on c_{ij}. A constraint network is constraint-based p-LC iff all values in all domains in D are constraint-based p-LC.*

Theorem 1. *Let LC be a local consistency stronger than AC for which the LC consistency of a value on a constraint is defined. Let p_1 and p_2 be two parameters in [0..1]. If $p_1 < p_2$ then AC $\preceq$ constraint-based p_1-LC $\preceq$ constraint-based p_2-LC $\preceq$ LC.*

Proof. Suppose that there exist two parameters p_1, p_2 such that $0 \leq p_1 < p_2 \leq 1$, and suppose that there exists a p_2-LC constraint network N that contains a p_2-LC value (x_i, v_i) that is p_1-LC inconsistent. Let c_{ij} be the constraint on which (x_i, v_i) is p_1-LC inconsistent. Then, $\nexists v_j \in D(x_j)$ that is an AC support for (x_i, v_i) on c_{ij} such that $\Delta(x_j, v_j) \geq p_1$. Thus, v_i is not p_2-stable for AC on c_{ij}. In addition, v_i is not LC on c_{ij}. Therefore, v_i is not p_2-LC, and N is not p_2-LC. ∎

Definition 3 can be modified to a more coarse-grained version that is not dependent on the consistency of values on a constraint. It will have the advantage to apply to any type of strong local consistency, even those, like singleton arc consistency, for which the consistency of a value on a constraint is not defined.

Definition 4 (Value-based p-LC). *Let LC be a local consistency stronger than AC. A value $v_i \in D(x_i)$ is value-based p-LC if and only if it is p-stable for AC or it is LC. A constraint network is value-based p-LC if and only if all values in all domains in D are value-based p-LC.*

Theorem 2. *Let LC be a local consistency stronger than AC. Let p_1 and p_2 be two parameters in [0..1]. If $p_1 < p_2$ then AC $\preceq$ value-based p_1-LC $\preceq$ value-based p_2-LC $\preceq$ LC.*

Proof. Suppose that there exist two parameters p_1, p_2 such that $0 \leq p_1 < p_2 \leq 1$, and suppose that there exists a p_2-LC constraint network N that contains a p_2-LC value (x_i, v_i) that is p_1-LC-inconsistent. v_i is p_1-LC-inconsistent means that:

1. v_i is not p_1-stable for AC: $\exists c_{ij}$ on which v_i is not p_1-stable for AC. Then $\nexists v_j \in D(x_j)$ that is an AC support for (x_i, v_i) on c_{ij} such that $\Delta(x_j, v_j) \geq p_1$. Therefore, v_i is not p_2-stable for AC on c_{ij}, then v_i is not p_2-stable for AC.
2. v_i is LC inconsistent

(1) and (2) imply that v_i is not p_2-LC, and N is not p_2-LC. ∎

For both types of definitions of p-LC, we have the following property on the extreme cases ($p = 0, p = 1$).

Corollary 1. *Let LC_1 and LC_2 be two local consistencies stronger than AC. We have: value-based 0-LC_2 = AC and value-based 1-LC_2 = LC. If the LC_1 consistency of a value on a constraint is defined, we also have: constraint-based 0-LC_1 = AC and constraint-based 1-LC_1 = LC.*

4 Parameterized maxRPC: p-maxRPC

To illustrate the benefit of our approach, we apply *parameterized consistency* to maxRPC to obtain the p-maxRPC level of consistency that achieves a consistency level between AC and maxRPC.

Definition 5 (p-maxRPC). *A value, a network, are p-maxRPC if and only if they are constraint-based p-maxRPC.*

From Theorem 1 and Corollary 1 we derive the following corollary.

Corollary 2. *For any two parameters p_1, $p_2, 0 \leq p_1 < p_2 \leq 1$, $AC \preceq p_1$-$maxRPC \preceq p_2$-$maxRPC \preceq maxRPC$. 0-$maxRPC = AC$ and 1-$maxRPC = maxRPC$.*

Algorithm 1: Initialization(X, D, C, Q)

```
1  begin
2      foreach x_i ∈ X do
3          foreach v_i ∈ D(x_i) do
4              foreach x_j ∈ Γ(x_i) do
5                  p-support ← false
6                  foreach v_j ∈ D(x_j) do
7                      if (v_i, v_j) ∈ c_ij then
8                          LastAC_{x_i,v_i,x_j} ← v_j
9                          if Δ(x_j, v_j) ≥ p then
10                             p-support ← true
11                             LastPC_{x_i,v_i,x_j} ← v_j
12                             break;
13                         if searchPCwit(v_i, v_j) then
14                             p-support ← true
15                             LastPC_{x_i,v_i,x_j} ← v_j
16                             break;
17                 if ¬p-support then
18                     remove v_i from D(x_i)
19                     Q ← Q ∪ {x_i}
20                     break;
21         if D(x_i) = ∅ then return false
22     return true
```

We propose an algorithm for p-maxRPC, based on maxRPC3, the best existing maxRPC algoritm. We do not describe maxRPC3 in full detail as it can be found in [1]. We only describe procedures where changes to maxRPC3 are

Algorithm 2: checkPCsupLoss(v_j, x_i)

```
1  begin
2    if LastAC_{x_j,v_j,x_i} ∈ D(x_i) then b_i ← max(LastPC_{x_j,v_j,x_i}+1, LastAC_{x_j,v_j,x_i})
3    else b_i ← max(LastPC_{x_j,v_j,x_i}+1, LastAC_{x_j,v_j,x_i}+1)
4    foreach v_i ∈ D(x_i), v_i ≥ b_i do
5      if (v_j, v_i) ∈ c_ji then
6        if LastAC_{x_j,v_j,x_i} ∉ D(x_i) & LastAC_{x_j,v_j,x_i} > LastPC_{x_j,v_j,x_i} then
7          LastAC_{x_j,v_j,x_i} ← v_i
8        if Δ(x_i, v_i) ≥ p then LastPC_{x_j,v_j,x_i} ← v_i return true
9        if searchPCwit(v_j, v_i) then LastPC_{x_j,v_j,x_i} ← v_i return true
10   return false
```

Algorithm 3: checkPCwitLoss(x_j, v_j, x_i)

```
1  begin
2    foreach x_k ∈ Γ(x_j) ∩ Γ(x_i) do
3      witness ← false
4      if v_k ← LastPC_{x_j,v_j,x_k} ∈ D(x_k) then
5        if Δ(x_k, v_k) ≥ p then witness ← true
6        else
7          if LastAC_{x_j,v_j,x_i} ∈ D(x_i) & LastAC_{x_j,v_j,x_i} = LastAC_{x_k,v_k,x_i}
8          OR LastAC_{x_j,v_j,x_i} ∈ D(x_i) & (LastAC_{x_j,v_j,x_i}, v_k) ∈ c_ik
9          OR LastAC_{x_k,v_k,x_i} ∈ D(x_i) & (LastAC_{x_k,v_k,x_i}, v_j) ∈ c_ij
10         then witness ← true
11         else
12           if searchACsup(x_j, v_j, x_i) & searchACsup(x_k, v_k, x_i) then
13             foreach v_i ∈ D(x_i), v_i ≥ max(LastAC_{x_j,v_j,x_i}, LastAC_{x_k,v_k,x_i})
               do
14               if (v_j, v_i) ∈ c_ji & (v_k, v_i) ∈ c_ki then
15                 witness ← true
16                 break;
17     if ¬witness & ¬checkPCsupLoss(v_j, x_k) then return false
18   return true
```

necessary to design p-maxRPC3, a coarse grained algorithm that performs p-maxRPC. We use light grey to emphasize the modified parts of the original maxRPC3 algorithm.

maxRPC3 uses a propagation list Q where it inserts the variables whose domains have changed. It also uses two other data structures: LastAC and LastPC. For each value (x_i, v_i) $\text{LastAC}_{x_i,v_i,x_j}$ stores the smallest AC support for (x_i, v_i) on c_{ij} and $\text{LastPC}_{x_i,v_i,x_j}$ stores the smallest PC support for (x_i, v_i) on c_{ij} (i.e.,

the smallest AC support (x_j, v_j) for (x_i, v_i) on c_{ij} such that (v_i, v_j) is PC). This algorithm consists in two phases: initialization and propagation.

In the initialization phase (algorithm 1) maxRPC3 checks if each value (x_i, v_i) has a maxRPC-support (x_j, v_j) on each constraint c_{ij}. If not, it removes v_i from $D(x_i)$ and inserts x_i in Q. To check if a value (x_i, v_i) has a maxRPC-support on a constraint c_{ij}, maxRPC3 looks first for an AC-support (x_j, v_j) for (x_i, v_i) on c_{ij}, then it checks if (v_i, v_j) is PC. In this last step, changes were necessary to obtain p-maxRPC3 (lines 9-12). We check if (v_i, v_j) is PC (line 13) only if $\Delta(x_j, v_j)$ is smaller than the parameter p (line 9).

The propagation phase of maxRPC3 consists in propagating the effect of deletions. While Q is non empty, maxRPC3 extracts a variable x_i from Q and checks for each value (x_j, v_j) of each neighboring variable $x_j \in \Gamma(x_i)$ if it is not maxRPC because of deletions of values in $D(x_i)$. A value (x_j, v_j) becomes maxRPC inconsistent in two cases: if its unique PC-support (x_i, v_i) on c_{ij} has been deleted, or if we deleted the unique witness (x_i, v_i) for a pair (v_j, v_k) such that (x_k, v_k) is the unique PC-support for (x_j, v_j) on c_{jk}. So, to propagate deletions, maxRPC3 checks if the last maxRPC support (last known support) of (x_j, v_j) on c_{ij} still belongs to the domain of x_i, otherwise it looks for the next support (algorithm 2). If such a support does not exist, it removes the value v_j and adds the variable x_j to Q. Then if (x_j, v_j) has not been removed in the previous step, maxRPC3 checks (algorithm 3) whether there is still a witness for each pair (v_j, v_k) such that (x_k, v_k) is the PC support for (x_j, v_j) on c_{jk}. If not, it looks for the next maxRPC support for (x_j, v_j) on c_{jk}. If such a support does not exist, it removes v_j from $D(x_j)$ and adds the variable x_j to Q.

In the propagation phase also, we modified maxRPC3 to check if the values are still p-maxRPC instead of checking if they are maxRPC. In p-maxRPC3, the last p-maxRPC support for (x_j, v_j) on c_{ij} is the last AC support if (x_j, v_j) is p-stable for AC on c_{ij}. If not, it is the last PC support. Thus, p-maxRPC3 checks if the last p-maxRPC support (last known support) of (x_j, v_j) on c_{ij} still belongs to the domain of x_i. If not, it looks (algorithm 2) for the next AC support (x_i, v_i) on c_{ij}, and checks if (v_i, v_j) is PC (line 9) only when $\Delta(x_i, v_i) < p$ (line 8). If no p-maxRPC support exists, p-maxRPC3 removes the value and adds the variable x_j to Q. If the value (x_j, v_j) has not been removed in the previous phase, p-maxRPC3 checks (algorithm 3) whether there is still a witness for each pair (v_j, v_k) such that (x_k, v_k) is the p-maxRPC support for v_j on c_{jk} and $\Delta(x_k, v_k) < p$. If not, it looks for the next p-maxRPC support for v_j on c_{jk}. If such a support does not exist, it removes v_j from $D(x_j)$ and adds the variable x_j to Q.

5 Experimental validation of *p*-maxRPC

To validate the approach of parameterized local consistency, we made a first basic experiment. The purpose of this experiment is to see if there exist problems on which a given level of p-maxRPC, with a p uniform on all the constraint network and static during the whole search is more efficient than AC or maxRPC, or both.

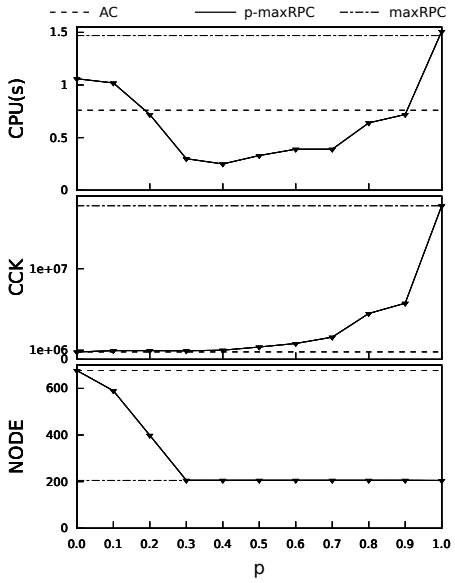

Fig. 2. Instance where p-maxRPC outperforms both AC and maxRPC.

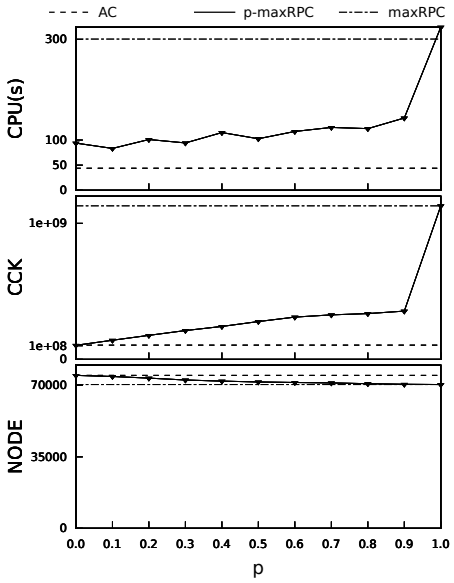

Fig. 3. Instance where AC outperforms p-maxRPC.

We have implemented the algorithms that achieve p-maxRPC as described in the previous section in our own binary constraint solver, in addition to maxRPC (maxRPC3 version [1]) and AC (AC2001 version [3]). All these algorithms are maintained during search. We tested these algorithms on several classes of problems of the International Constraint Solver Competition 09[1]. We have only selected problems involving binary constraints. To isolate the effect of propagation, we used the lexicographic ordering for variables and values. We set the CPU timeout to one hour. Our experiments were conducted on a 12-core Genuine Intel machine with 16Gb of RAM running at 2.92GHz.

On each instance of our experiment, we ran AC, max-RPC, and p-maxRPC for all values of p in $\{0.1, 0.2, \ldots, 0.9\}$. Performance has been measured in terms of CPU time in seconds, the number of visited nodes (NODE) and the number of constraint checks (CCK). Results are presented as "CPU time (p)", where p is the parameter for which p-maxRPC gives the best result.

Table 1 reports the performance of AC, maxRPC, and p-maxRPC for the value of p producing the best CPU time, on Radio Link Frequency Assignment Problems (RLFAPs), Geom problems, and queens knights problems. The CPU time of the best algorithm is highlighted with bold. On RLFAP and Geom, we observe the existence of a parameter p where p-maxRPC is faster than *both* AC and maxRPC for most instances of these two classes of problems. On the queens-knight problem, however, AC is always the best algorithm. In Figures 2 and 3, we

[1] http://cpai.ucc.ie/09/

Table 1. Performance (CPU time, nodes and constraint checks) of AC, p-maxRPC, and maxRPC on various instances.

		AC	p-maxRPC	maxRPC
scen1-f8	CPU(s)	Time-out	**1.39** (0.2)	6.10
	#nodes	–	927	917
	#ccks	–	1,397,440	26,932,990
scen2-f24	CPU(s)	Time-out	**0.13** (0.3)	0.65
	#nodes	–	201	201
	#ccks	–	296,974	3,462,070
scen3-f10	CPU(s)	Time-out	**0.89** (0.5)	2.80
	#nodes	–	469	408
	#ccks	–	874,930	13,311,797
geo50-20-d4-75-26	CPU(s)	111.48	17.80 (1.0)	**15.07**
	#nodes	477,696	3,768	3,768
	#ccks	96,192,822	40,784,017	40,784,017
geo50-20-d4-75-43	CPU(s)	1,671.35	**1,264.36** (0.5)	1,530.02
	#nodes	4,118,134	555,259	279,130
	#ccks	1,160,664,461	1,801,402,535	3,898,964,831
geo50-20-d4-75-46	CPU(s)	1,732.22	**371.30** (0.6)	517.35
	#nodes	3,682,394	125,151	64,138
	#ccks	1,516,856,615	584,743,023	1,287,674,430
geo50-20-d4-75-84	CPU(s)	404.63	**0.44** (0.6)	0.56
	#nodes	2,581,794	513	333
	#ccks	293,092,144	800,657	1,606,047
queensKnights10-5-add	CPU(s)	**27.14**	30.79 (0.2)	98.44
	#nodes	82,208	81,033	78,498
	#ccks	131,098,933	148,919,686	954,982,880
queensKnights10-5-mul	CPU(s)	**43.89**	83.27 (0.1)	300.74
	#nodes	74,968	74,414	70,474
	#ccks	104,376,698	140,309,576	1,128,564,278

try to understand more closely what makes p-maxRPC better or worse than AC and maxRPC. Figures 2 and 3 plot the performance (CPU, NODE and CCK) of p-maxRPC for all values of p from 0 to 1 by steps of 0.1 against performance of AC and maxRPC. Figure 2 shows an instance where p-maxRPC solves the problem faster than AC and maxRPC for values of p in the range [0.3..0.8]. We observe that p-maxRPC is faster than AC and maxRPC when it reduces the size of the search space as much as maxRPC (same number of nodes visited) with a number of CCK closer to the number of CCK produced by AC. Figure 3 shows an instance where the CPU time for p-maxRPC is never better than *both* AC and maxRPC. We see that if the CPU time for p-maxRPC is two to three times better than maxRPC, it fails to improve AC because the number of constraint checks performed by p-maxRPC is much higher than the number of constraint checks performed by AC, whereas the number of nodes visited by p-maxRPC is not significantly reduced compared to the number of nodes visited by AC. From

these observations, it thus seems that p-maxRPC outperforms AC and maxRPC when it finds a compromise between the number of nodes visited (the power of maxRPC) and the number of CCK needed to maintain (the light cost of AC).

In Figures 2 and 3 we can see that the CPU time for 1-maxRPC (respectively 0-maxRPC) is greater than the CPU time for maxRPC (respectively AC) although the two consistencies are equivalent. The reason is that p-maxRPC performs tests on the distances. For $p = 0$, we also explain this difference by the fact that p-maxRPC maintains data structures that AC does not use.

6 Adaptative Parameterized Consistency: *ap*-maxRPC

In the previous section, we have defined p-maxRPC, a version of parameterized consistency where the strong local consistency is maxRPC. We have performed some initial experiments where p has the same value during the whole search and everywhere in the constraint network. However, the algorithm we proposed to enforce p-maxRPC does not specify how p is chosen. In this section, we propose two possible ways to dynamically and locally adapt the parameter p in order to solve the problem faster than both AC and maxRPC. Instead of using a single parameter p during the whole search and for the whole constraint network, we propose to use several local parameters and to adapt the level of local consistency by dynamically adjusting the value of the different local parameters during search. The idea is to concentrate the effort of propagation by increasing the level of consistency in the most difficult parts of the instance. We can determine these difficult parts using heuristics based on conflicts in the same vein as the weight of a constraint or the weighted degree of a variable in [5].

6.1 Constraint-Based *ap*-maxRPC : *apc*-maxRPC

The first technique we propose, called constraint-based ap-maxRPC, assigns a parameter $p(c_k)$ to each constraint c_k in C. We define this parameter to be correlated to the *weight* of the constraint. The idea is to apply a higher level of consistency in parts of the problem where the constraints are the most active.

Definition 6 (The weight of a constraint [5]). *The weight $w(c_k)$ of a constraint $c_k \in C$ is an integer that is incremented every time a domain wipe out occurs while performing propagation on this constraint.*

We define the adaptive parameter $p(c_k)$ local to constraint c_k in such a way that it is greater when the weight $w(c_k)$ is higher wrt to other constraints.

$$\forall c_k \in C, p(c_k) = \frac{w(c_k) - min_{c \in C}(w(c))}{max_{c \in C}(w(c)) - min_{c \in C}(w(c))} \quad (1)$$

Equation 1 is normalized so that we are guaranteed that $0 \leq p(c_k) \leq 1$ for all $c_k \in C$ and that there exists c_{k_1} with $p(c_{k_1}) = 0$ (the constraint with lowest weight) and c_{k_2} with $p(c_{k_2}) = 1$ (the constraint with highest weight).

We are ready to define adaptive parameterized consistency based on constraints.

Definition 7 (constraint-based *ap*-maxRPC). *A value $v_i \in D(x_i)$ is constraint-based ap-maxRPC (or apc-maxRPC) on a constraint c_{ij} if and only if it is constraint-based $p(c_{ij})$-maxRPC. A value $v_i \in D(x_i)$ is apc-maxRPC iff $\forall c_{ij}$, v_i is apc-maxRPC on c_{ij}. A constraint network is apc-maxRPC iff all values in all domains in D are apc-maxRPC.*

6.2 Variable-Based *ap*-maxRPC: *apx*-maxRPC

The technique proposed in Section 6.1 can only be used on consistencies where the consistency of a value on a constraint is defined. We give a second technique which can be used on constraint-based or variable-based local consistencies indifferently. We instantiate our definitions to maxRPC but the extension to other consistencies is direct. We call this new technique variable-based *ap*-maxRPC. We need to define the weighted degree of a variable as the aggregation of the weights of all constraints involving it.

Definition 8 (The weighted degree of a variable [5]). *The weighted degree $wdeg(x_i)$ of a variable x_i is the sum of the weights of the constraints involving x_i and one other uninstantiated variable.*

We associate each variable with an adaptive local parameter based on its weighted degree.

$$\forall x_i \in X, p(x_i) = \frac{wdeg(x_i) - min_{x \in X}(wdeg(x))}{max_{x \in X}(wdeg(x)) - min_{x \in X}(wdeg(x))} \quad (2)$$

As in Equation 1, we see that the local parameter is normalized so that we are guaranteed that $0 \leq p(x_i) \leq 1$ for all $x_i \in X$ and that there exists x_{k_1} with $p(x_{k_1}) = 0$ (the variable with lowest weighted degree) and x_{k_2} with $p(x_{k_2}) = 1$ (the variable with highest weighted degree).

Definition 9 (variable-based *ap*-maxRPC). *A value $v_i \in D(x_i)$ is variable-based ap-maxRPC (or apx-maxRPC) if and only if it is value-based $p(x_i)$-maxRPC. A constraint network is apx-maxRPC iff all values in all domains in D are apx-maxRPC.*

7 Experimental Evaluation of *ap*-maxRPC

In Section 5 we have shown that maintaining a static form of p-maxRPC during the whole search can lead to a promising trade-off between computational effort and pruning when all algorithms follow the same static variable ordering. In this section, we want to put our contributions in the real context of a solver using the best known variable ordering heuristic, *dom/wdeg*, though it is known that this heuristic is so good that it reduces a lot the differences in performance that other features of the solver could provide. We have compared the two variants of adaptive parameterized consistency introduced in Section 6 to AC and maxRPC.

We ran the four algorithms on radio link frequency assignment problems, geom problems, and queens knights problems.

Table 2 reports some representative results. A first observation is that, thanks to the *dom/wdeg* heuristic, we were able to solve more instances before the cutoff of one hour, especially the scen11 variants of RLFAP. A second observation is that *apc*-maxRPC and *apx*-maxRPC are both faster than at least one of the two extreme consistencies (AC and maxRPC) on all instances except scen7-w1-f4 and geo50-20-d4-75-30. Third, when *apx*-maxRPC and/or *apc*-maxRPC are faster than both AC and maxRPC (scen1-f9, scen2-f25, scen11-f9, scen11-f10 and scen11-f11), we observe that the gap in performance in terms of nodes and CCKs between AC and maxRPC is significant. Except for scen7-w1-f4, the number of nodes visited by AC is three to five times greater than the number of nodes visited by maxRPC and the number of constraint checks performed by maxRPC is twelve to sixteen times greater than the number of constraint checks performed by AC. For the Geom instances the CPU time of the *ap*-maxRPC algorithms is between AC and maxRPC, and it is never lower than the CPU time of AC. This probably means that when solving these instances with the *dom/wdeg* heuristic, there is no need for sophisticated local consistencies. In general we see that the *ap*-maxRPC algorithms fail to improve both the two extreme consistencies simultaneously for the instances where the performance gap between AC and maxRPC is low.

If we compare *apx*-maxRPC to *apc*-maxRPC, we observe that although *apx*-maxRPC is coarser in its design than *apc*-maxRPC, *apx*-maxRPC is often faster than *apc*-maxRPC. We can explain this by the fact that the constraints initially all have the same weight equal to 1. Hence, all local parameters $ap(c_k)$ initially have the same value 0, so that *apc*-maxRPC starts resolution by applying AC everywhere. It will start enforcing some amount of maxRPC only after the first wipe-out occurred. On the contrary, in *apx*-maxRPC, when constraints all have the same weight, the local parameter $p(x_i)$ is correlated to the degree of the variable x_i. As a result, *apx*-maxRPC benefits from the filtering power of maxRPC even before the first wipe-out.

In Table 2, we reported only the results on a few representative instances. Table 3 summarizes the whole set of experiments. It shows the average CPU time for each algorithm on all instances of the different classes of problems tested. We considered only the instances solved before the cutoff of one hour by at least one of the four algorithms. To compute the average CPU time of an algorithm on a class of problems, we add the CPU time needed to solve each instance solved before the cutoff of one hour, and for the instances not solved before the cutoff, we add one hour. We observe that the adaptive approach is, on average, faster than the two extreme consistencies AC and maxRPC, except on the Geom class.

In *apx*-maxRPC and *apc*-maxRPC, we update the local parameters $p(x_i)$ or $p(c_k)$ at each node in the search tree. We could wonder if such a frequent update does not produce too much overhead. To answer this question we performed a simple experiment in which we update the local parameters every 10 nodes only. We re-ran the whole set of experiments with this new setting. Table 4 reports

Table 2. Performance (CPU time, nodes and constraint checks) of AC, variable-based *ap*-maxRPC (*apx*-maxRPC), constraint-based *ap*-maxRPC (*apc*-maxRPC), and maxRPC on various instances.

		AC	***apx*-maxRPC**	***apc*-maxRPC**	**maxRPC**
scen1-f9	CPU(s)	90.34	**31.17**	33.40	41.56
	#nodes	2,291	1,080	1,241	726
	#ccks	3,740,502	3,567,369	2,340,417	50,045,838
scen2-f25	CPU(s)	70.57	46.40	**27.22**	81.40
	#nodes	12,591	4,688	3,928	3,002
	#ccks	15,116,992	38,239,829	8,796,638	194,909,585
scen6-w2	CPU(s)	7.30	1.25	2.63	**0.01**
	#nodes	2,045	249	610	0
	#ccks	2,401,057	1,708,812	1,914,113	85,769
scen7-w1-f4	CPU(s)	0.28	**0.17**	0.54	0.30
	#nodes	567	430	523	424
	#ccks	608,040	623,258	584,308	1,345,473
scen11-f9	CPU(s)	2,718.65	**1,110.80**	1,552.20	2,005.61
	#ccks	103,506	40,413	61,292	32,882
	#nodes	227,751,301	399,396,873	123,984,968	3,637,652,122
scen11-f10	CPU(s)	225.29	**83.89**	134.46	112.18
	#ccks	9,511	3,510	4,642	2,298
	#nodes	12,972,427	17,778,458	6,717,485	156,005,235
scen11-f11	CPU(s)	156.76	**39.39**	93.69	76.95
	#ccks	7,050	2,154	3,431	1,337
	#nodes	7,840,552	10,006,821	5,143,592	91,518,348
scen11-f12	CPU(s)	139.91	69.50	88.76	**61.92**
	#ccks	7,050	2,597	3,424	1,337
	#nodes	7,827,974	11,327,536	5,144,835	91,288,023
geo50-20d4-75-19	CPU(s)	**242.13**	553.53	657.72	982.34
	#nodes	195,058	114,065	160,826	71,896
	#ccks	224,671,319	594,514,132	507,131,322	2,669,750,690
geo50-20d4-75-30	CPU(s)	**0.84**	1.01	1.07	1.02
	#nodes	359	115	278	98
	#ccks	261,029	432,705	313,168	1,880,927
geo50-20d4-75-84	CPU(s)	**0.02**	0.09	0.05	0.29
	#nodes	59	54	59	52
	#ccks	33,876	80,626	32,878	697,706
queensK20-5-mul	CPU(s)	787.35	2,345.43	**709.45**	Time-out
	#codes	55,596	40,606	41,743	–
	#ccks	347,596,389	6,875,941,876	379,826,516	–
queensK15-5-add	CPU(s)	24.69	17.01	**14.98**	35.05
	#codes	24,639	12,905	12,677	11,595
	#ccks	90,439,795	91,562,150	58,225,434	394,073,525

Table 3. Average CPU time of AC, variable-based *ap*-maxRPC (*apx*-maxRPC), constraint-based *ap*-maxRPC (*apc*-maxRPC), and maxRPC on all instances of each class of problems tested, when the local parameters are updated at each node

		AC	***apx*-maxRPC**	***apc*-maxRPC**	**maxRPC**
Average(CPU)	**geom**	**69.28**	180.57	191.03	279.30
	scen	18.95	9.63	**8.30**	13.94
	scen11	810.15	**325.90**	467.28	564.17
	queensK	135.95	395.41	**121.75**	610.51

Table 4. Average CPU time of AC, variable-based *ap*-maxRPC (*apx*-maxRPC), constraint-based *ap*-maxRPC (*apc*-maxRPC), and maxRPC on all instances of each class of problems tested, when the local parameters are updated every 10 nodes

		AC	***apx*-maxRPC**	***apc*-maxRPC**	**maxRPC**
Average(CPU)	**geom**	**69.28**	147.20	189.42	279.30
	scen	18.95	**7.40**	8.86	13.94
	scen11	810.15	**311.74**	417.97	564.17
	queensK	135.95	269.51	**117.18**	610.52

the CPU time average results. We observe that when the local parameters are updated every 10 nodes, the gain for the adaptive approach is, on average, greater than when the local parameters are updated at each node. This gives room for improvement, by trying to adapt the frequency of update of these parameters.

8 Conclusion

We have introduced the notion of stability of values for arc consistency, a notion based on the depth of their supports in their domain. We have used this notion to propose parameterized consistency, a technique that allows to define levels of local consistency of increasing strength between arc consistency and a given strong local consistency. We have instantiated the generic parameterized consistency approach to max restricted path consistency. We have experimentally shown that the concept of parameterized consistency is viable. Then we have introduced two techniques which allow us to make the parameter adaptable dynamically and locally during search . We have evaluated these two techniques experimentally and we have observed that adapting the level of local consistency during search using the parameterized consistency concept is a promising approach that can outperform both MAC and a strong local consistency on many problems.

References

1. Balafoutis, T., Paparrizou, A., Stergiou, K., Walsh, T.: New algorithms for max restricted path consistency. Constraints 16(4), 372–406 (2011)
2. Bessiere, C.: Constraint propagation. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, chap. 3. Elsevier (2006)
3. Bessiere, C., Régin, J.C., Yap, R.H.C., Zhang, Y.: An optimal coarse-grained arc consistency algorithm. Artif. Intell. 165(2), 165–185 (2005)
4. Bessiere, C., Stergiou, K., Walsh, T.: Domain filtering consistencies for non-binary constraints. Artif. Intell. 172(6-7), 800–822 (2008)
5. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: ECAI. pp. 146–150 (2004)
6. Debruyne, R., Bessiere, C.: Domain filtering consistencies. Journal of Artificial Intelligence Research 14, 205–230 (2001)
7. Debruyne, R., Bessière, C.: Some practicable filtering techniques for the constraint satisfaction problem. In: IJCAI (1). pp. 412–417 (1997)
8. Katriel, I., Van Hentenryck, P.: Randomized filtering algorithms. Technical Report CS-06-09, Brown University (June 2006)
9. Sellmann, M.: Approximated consistency for knapsack constraints. In: CP. pp. 679–693 (2003)
10. Stergiou, K.: Heuristics for dynamically adapting propagation in constraint satisfaction problems. AI Commun. 22, 125–141 (August 2009)