Singleton consistency and dominance for weighted CSP

Simon de Givry

Laboratoire de Biométrie et Intelligence Artificielle
INRA, Toulouse, France
simon.degivry@toulouse.inra.fr

Abstract. The weighted CSP (WCSP) framework is a soft constraint framework with a wide range of applications. In this paper, we propose an improved singleton consistency algorithm for solving WCSP. By exploiting the singleton test when it does not fail, we can prune more values and we can produce better lower bounds. These improvements are based on constructive disjunction and subproblem aggregation. Experimental results showed that a restricted version (which does not perform all the singleton tests) of our algorithm can outperforms a state-of-the-art WCSP solver on some random and structured problems. In the hope of further restricting the search space, we also study interesting dominance rules and show their relation with soft substitutability.

1 Introduction

In the last few years, the CSP framework has been augmented with so-called *soft constraints* which allows to express preferences among solutions [22, 4]. Soft constraint frameworks associate costs to tuples and the goal is to find a complete assignment with minimum combined cost. *Weighted Constraint Satisfaction Problems* (WCSP) is a well known soft constraint framework with a non-idempotent cost aggregation operator $\oplus$ (Max-CSP [14], Max-SAT [7], and Bayesian Networks [21] can be expressed as WCSPs) and with a wide range of applications. Enforcing local consistency properties during the search plays a major role in CSP solving and this has been adapted to solve WCSP by Larrosa and Schiex [13, 14]. In [14], experiments on random binary Max-CSPs show that the best results are obtained by the local consistency *FDAC** which has the highest worst-case time complexity among the studied local consistencies. This result naturally raises the possible utility of considering more complex local consistencies in order to get better results. In this paper, we propose a local consistency stronger than FDAC* based on singleton consistency and we show it can be worthwhile for some random and structured WCSPs. Singleton Arc Consistency (SAC) has been proposed first by Debruyne and Bessière [8] in the CSP framework, further studied in Prosser *et al* [19]. For each variable, for each value, SAC performs a singleton test, *i.e.* assigns this value to this variable, then it applies arc consistency to detect if this value makes the problem infeasible and can be safely removed in the original problem ; this process is iterated until no value can be removed. Singleton consistency is a meta-level local consistency that can be applied to any local consistency, *e.g.* arc consistency. The direct application of singleton consistency to WCSP consists in removing values whose singleton test produces a lower bound greater than the current upper bound. We propose two new improvements for singleton consistency, the first one based on constructive disjunction applicable to both CSP and WCSP frameworks, and the second one based on subproblem aggregation which further improves the lower bound of WCSP. We give theoretical results showing the relation between the different local consistencies. In the hope of further restricting the search space, we also study interesting dominance rules introduced by Koster [12] and show their relation with soft substitutability defined by Bistarelli *et al* [3]. Experiments on random and structured WCSPs are given, showing the efficiency and complementarity of singleton consistency and dominance.

2 Weighted CSP and local consistencies

As defined in [13, 14], a weighted CSP (WCSP) $\mathcal{P}$ is a quadruplet $(\mathcal{X}, \mathcal{D}, \mathcal{C}, k)$. $\mathcal{X} = \{1, \ldots, n\}$ is a set of variables. Each variable $i \in \mathcal{X}$ has a finite domain $D_i \in \mathcal{D}$ of values than can be assigned to it. The maximum domain size is d. $\mathcal{C}$ is a set of e constraints which are actually cost functions. All the cost functions return a natural number lower than or equal to $k \in [1, \infty]$. The combination of two costs is done using bounded addition denoted $\oplus$ and defined as $a \oplus b = \min\{k, a + b\}$. The value k is an absorbing element ($\alpha \oplus k = k = \top$) and corresponds to forbidden tuples. For instance, $C_{i,j} \in \mathcal{C}$ is a binary cost function applied on variables i and j. We have $\forall (a, b) \in D_i \times D_j, C_{i,j}(a, b) \in [0, k]$. We consider only binary WCSPs in the sequel. Without changing the problem semantics, we assume the existence of a unary cost function for every variable. Missing cost functions initially return zero. A zero arity constraint $C_\emptyset$ is also introduced, which initially returns zero. $C_\emptyset$ will be used by local consistencies to store the current problem lower bound. The goal is to find a *complete consistent assignment with minimum cost*: $\min_{(v_1,v_2,\ldots,v_n)\in D_1\times D_2\times\cdots\times D_n}(C_\emptyset \oplus \sum_{i\in\mathcal{X}}(C_i(v_i)) \oplus \sum_{(i,j)\in\mathcal{X}\times\mathcal{X},i<j}(C_{i,j}(v_i, v_j)))$. An assignment is *consistent* if its cost is strictly lower than k. In general, this combinatorial optimization problem is NP-hard.

In this paper, we are interested in the following local consistencies defined in [20, 13, 6, 14]:

- Node consistency (NC*). A problem is NC* if $\forall i \in \mathcal{X}, \forall a \in D_i, C_\emptyset \oplus C_i(a) < k$ and $\forall i \in \mathcal{X}, \exists a \in D_i \;\; C_i(a) = 0$. Value a is called a *support* of variable i.
- Arc consistency (AC*). A problem is AC* if it is NC* and $\forall i \in \mathcal{X}, \forall j \in \mathcal{X}, i \neq j, \forall a \in D_i, \exists b \in D_j \;\; C_{i,j}(a, b) = 0$. Value b is called a *support* of value a.
- Directional arc consistency (DAC*). A problem is DAC* if it is NC* and $\forall i \in \mathcal{X}, \forall j \in \mathcal{X}, i < j, \forall a \in D_i, \exists b \in D_j \;\; C_{i,j}(a, b) + C_j(b) = 0$. Value b is called a *full support* of value a.
- Full directional arc consistency (FDAC*). A problem is FDAC* if it is AC* and DAC*.

Two WCSPs defined over the same variables are said to be *equivalent* if they define the same cost distribution on complete assignments. Enforcing local consistency on a given problem is done by changing the values returned by its cost functions. For that, cost functions are implemented as tables which can be consulted and modified. NC* will remove forbidden values and *project* C_i to $C_\emptyset$ in order to get a support for i. AC* will *project* $C_{i,j}$ to C_i in order to get a support for each value in D_i. DAC* will first *extend* C_j to $C_{i,j}$ and then *project* $C_{i,j}$ to C_i. *Projection* and *extension* are the only two basic operations used to transform a given problem into an equivalent but possibly more explicit one (having a better lower bound $C_\emptyset$ and less values). See [14] for a definition and examples of projection and extension operations. An important result of enforcing local consistency is to produce a problem lower bound in $C_\emptyset$ that will be used to restrict the search space. In [14], local consistency enforcement algorithms are proposed. Their worst-case time complexities are in increasing order of complexity $O(nd)$ for NC*, $O(ed^2)$ for DAC*, $O(n^2d^3)$ for AC*, and $O(end^3)$ for FDAC*. The space complexity is $O(ed)$ for every local consistency. FDAC* is *strictly stronger* (see the definition in Section 3.3) than AC* and DAC*. AC* and DAC* are incomparable. NC* is the weakest consistency. Each local consistency can be maintained during the search as explained in [14]. The Depth First Branch and Bound principle is applied using k as the current upper bound (k is equal to the cost of the best solution found so far) and $C_\emptyset$ as the current lower bound.

3 Singleton Consistencies

The generic singleton consistency algorithm as defined in [8] is described in Algorithm 1. $\mathcal{P}_{|D_i=\{a\}}$ represents the subproblem of $\mathcal{P}$ where value a is assigned to variable i. The local consistency function Singleton($\mathcal{P}$,Lc) takes two parameters as inputs: the current problem $\mathcal{P}$ and the inherited local consistency Lc which is applied on $\mathcal{P}$ and $\mathcal{P}_{|D_i=\{a\}}$[1]. A local consistency function transforms its input problem into a simpler one and returns **true** if no domain wipe out occurs, or equivalently, if the deduced lower bound is strictly lower than the current upper bound. The worst-case time complexity of Singleton($\mathcal{P}$,Lc) is $O(n^2d^2)$ times the time complexity of Lc() (when a single value is removed at each step of the **repeat** loop). The space complexity of Singleton($\mathcal{P}$,Lc) is the same as for Lc(). When FDAC* is used as the inherited local consistency, the complexity of Singleton($\mathcal{P}$,FDAC*) is time $O(en^3d^5)$ and space $O(ed)$.

3.1 Exploiting the singleton test when it does not fail

In this section, we show how to enhance the initial singleton algorithm by exploiting the singleton test when it does not fail.

Constructive disjunction. The first idea is to apply the constructive disjunction principle initially proposed in [25] for the Constraint Logic Programming framework and further studied in [11, 26]. Informally, any value which is forbidden in all the successful singleton tests for a given variable can be removed in the original problem. Formally expressed, when Lc($\mathcal{P}_{|D_i=\{a\}}$) returns **true**, we keep the set $Z_{i,a}$ of values that are forbidden in the subproblem $\mathcal{P}_{|D_i=\{a\}}$ after enforcing local consistency. Now, we can deduce that all values in $\bigcap_{a \in D_i / Lc(\mathcal{P}_{|D_i=\{a\}})=true}(Z_{i,a})$ can be safely removed from $\mathcal{P}$.

[1] One might think using different levels of local consistency for both problems ; in particular, a simpler one for $\mathcal{P}_{|D_i=\{a\}}$ because it is called very frequently.

```
Function Singleton(P,Lc) : boolean
    if ¬ Lc(P) then return false;
    repeat
        pruned := false;
        foreach i ∈ X do
            foreach a ∈ D_i do
                if ¬ Lc(P_{|D_i={a}}) then
                    D_i := D_i − {a};
                    pruned := true;
                    if ¬Lc(P) then return false;
    until ¬pruned;
    return true;
```

Algorithm 1: Singleton consistency

In order to speed up this new filtering process, and in particular to compute the set intersection efficiently, we use two global data structures: Z is a set of removed values and $R(j, b)$ is an array which stores the number of times value (j, b) is found forbidden by a successful singleton test for a given variable. Z is reset before testing each value and R is reset at line 4 of Algorithm 2 before testing each variable. A local variable *success* counts the number of successful singleton tests for a given variable. For each singleton test, a modified Lc, called Lc', is enforced at line 5. The modification is that when Lc' removes a value (j, b), it inserts this value in Z if and only if (j, b) was also removed in all the previous successful singleton tests (line 9 in procedure PruneValue). When the singleton test succeeds, Z contains only removed values which have been also removed by the previous successful singleton tests and R is updated at line 6 to ensure that $R(j, b)$ counts the number of times value (j, b) is found forbidden by a successful singleton test. Moreover, we store a copy of Z in Z_{prev}[2]. When the singleton test fails, the information in Z is useless. After testing a variable, we remove at line 7 all the values belonging to Z_{prev} which contains the set of the last successful singleton test. By definition of PruneValue, this set contains only the relevant values. When this set is not empty, we enforce Lc again at line 8. Notice that the practical complexity of the loops at line 6 and 7 rapidly converges (after the first singleton test) to the size of the last successful singleton test set. The resulting algorithm, called Singleton$_{cdisj}$(), is given in Algorithm 2.

In [18, 17], the link between constructive disjunction and filtering based on arc consistency counts [9] is presented, but it is not embodied in singleton consistency.

Improving the lower bound. We describe now our second improvement on singleton consistency for WCSPs. By looking at the lower bound obtained at each successful singleton test (recall that enforcing local consistency produces a lower bound in $C_\emptyset$),

[2] This copy can be avoided by using a list vector instead of Z, containing the list of removed values for each successful singleton test.

```
Function Singleton_cdisj(P,Lc) : boolean
    if ¬ Lc(P) then return false;
    repeat
        pruned := false;
        foreach i ∈ X do
            success := 0 ;
            Zprev := ∅ ;
            foreach j ∈ X do foreach b ∈ D_j do R(j, b) := 0;
            foreach a ∈ D_i do
                Z := ∅ ;
                if ¬ Lc'(P_{|D_i={a}}) then
                    D_i := D_i − {a};
                    pruned := true;
                    if ¬Lc(P) then return false;
                else
                    foreach (j, b) ∈ Z do R(j, b) := R(j, b) + 1 ;
                    success := success + 1;
                    Zprev := Z ;
            if Zprev ≠ ∅ then
                foreach (j, b) ∈ Zprev do D_j := D_j − {b};
                pruned := true;
                if ¬Lc(P) then return false;
    until ¬pruned;
    return true;

Procedure PruneValue(i,a) /* called by Lc' each time value a is removed from D_i */
    D_i := D_i − {a};
    if R(i, a) = success then Z := Z ∪ {(i, a)};
```

Algorithm 2: Singleton consistency with constructive disjunction we can simply derive a new lower bound for $\mathcal{P}$ equal to:

$$\max_{i \in \mathcal{X}} (\min_{a \in D_i} (C_\emptyset^{\mathcal{P}_{|D_i=\{a\}}}))$$

Unfortunately, this lower bound is by definition strictly lower than the current upper bound and it cannot be simply integrated in $C_\emptyset^{\mathcal{P}}$ to make the problem $\mathcal{P}$ more explicit.

In some particular cases, it is however possible to transform $\mathcal{P}$ into an equivalent but more explicit problem $\mathcal{P}'$ by the aggregation of subproblems $\mathcal{P}_{|D_i=\{a\}} \forall a \in D_i$. The resulting problem $\mathcal{P}'$ has $C_\emptyset^{\mathcal{P}'} \geq \min_{i \in \mathcal{X}}(\min_{a \in D_i}(C_\emptyset^{\mathcal{P}_{|D_i=\{a\}}}))$. In particular, when a better lower bound $C_\emptyset^{\mathcal{P}_{|D_i=\{a\}}} > C_\emptyset^{\mathcal{P}}$ is found for each successful singleton test of a given variable i without modifying any binary cost $C_{j,k}$ such that $j \neq k \neq i$, then $C_\emptyset^{\mathcal{P}'} = \min_{a \in D_i}(C_\emptyset^{\mathcal{P}_{|D_i=\{a\}}})$.

Let see an example in Fig. 1. The original problem $\mathcal{P}$ has 3 binary variables and is described in Fig. 1.a. Unary costs are depicted inside their domain value. Binary costs are depicted as labelled edges connecting the corresponding pair of values (default cost of 1). Zero costs are not shown. When applying Singleton(NC*) on variable z, the resulting subproblems are given in Fig. 1.b and Fig. 1.c. $\mathcal{P}_{|D_z=\{a\}}$ and $\mathcal{P}_{|D_z=\{b\}}$ have both a lower bound of 1. The only differences between Fig. 1.b and Fig. 1.c are: (i) the unary costs $C_x(a)$ and $C_y(a)$ (ii) the unary costs $C_z(a)$ and $C_z(b)$. In order to merge common information between both figures, we extend the unary costs $C_y(a)$ and $C_x(a)$ on the binary costs $C_{y,z}(a,a)$ and $C_{x,z}(a,b)$ respectively, producing Fig. 1.d and Fig. 1.e. Now, the only differences are directly related to the variable z. Thus, we can merge these two figures safely, producing an equivalent problem $\mathcal{P}'$ given in Fig. 1.e with a lower bound $C_\emptyset^{\mathcal{P}'} = 1$.

Suppose there is a difference in terms of lower bound between the singleton tests. In this case, $C_\emptyset^{\mathcal{P}'}$ is equal to the minimum lower bound and each absolute difference $C_\emptyset^{\mathcal{P}_{|D_i=\{a\}}} - C_\emptyset^{\mathcal{P}'}$ is moved on the corresponding unary cost $C_i(a)$ to compensate this difference. But if there is a difference on any binary cost not directly related to the variable currently examined by Singleton(), then merging the singleton tests may imply the addition of ternary cost functions. This is why we limit the aggregation procedure to the case where no binary cost changes.

Notice that if we apply DAC* with the variable ordering $z < y < x$ to the problem in Fig. 1.a, then it also obtains a lower bound of 1. The subproblem aggregation technique allows us to derive a new variable ordering heuristic for DAC*. When singleton consistency discovers that a subproblem aggregation is possible for a given variable, it temporally moves this variable first in the DAC* variable ordering, then it enforces DAC* for this variable. In this case, it performs exactly what subproblem aggregation does. Testing for subproblem aggregation can simply be done by checking if the lower bound has been increased after enforcing NC* in each singleton test. We suppose that in the implementation of DAC* and FDAC*, NC* is enforced before enforcing DAC or FDAC. In conclusion, this method, called Singleton$_{lb}$() in the experiments, can detect at no extra cost when it is interesting to try a different variable ordering for DAC* used in Singleton$_{lb}$(DAC*) or Singleton$_{lb}$(FDAC*).

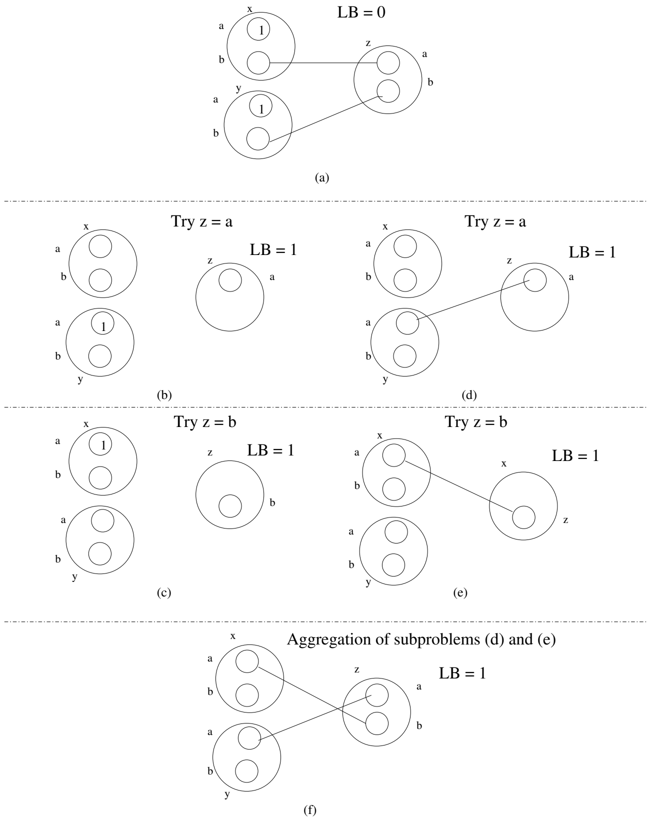

Fig. 1. (a) Original problem $\mathcal{P}$. (b) Singleton test $\mathcal{P}_{|D_z=\{a\}}$ using NC*. (d) Equivalent problem $\mathcal{P}_{|D_z=\{a\}}$. (c) Singleton test $\mathcal{P}_{|D_z=\{b\}}$ using NC*. (e) Equivalent problem $\mathcal{P}_{|D_z=\{b\}}$. (f) Aggregation of $\mathcal{P}_{|D_z=\{a\}}$ and $\mathcal{P}_{|D_z=\{b\}}$. $k = \top = 3$.

3.2 Restricted forms of singleton consistency

Because of its high time complexity, restricted forms of singleton consistency have been proposed in the literature. In the CSP framework [19], a restricted singleton consistency is defined by removing the **repeat** loop in line 1. In the experiments, we add the symbol ^{iter}, e.g. $\mathsf{Singleton}^{iter}$, when the **repeat** loop is used. A more drastic approach is to restrict the number of examined variables in line 2 to the most constrained ones. For 3-SAT problems [15], the variables having a large number of binary occurrences (i.e occurring in binary clauses) are examined. In our experiments, the variable ordering heuristic for choosing the next variable to branch in the search tree is also used for choosing the most constrained variables for singleton consistency. The number of examined variables is given as a percentage. For instance, Singleton(25,FDAC*) applies Singleton(FDAC*) on $\frac{25n}{100}$ unassigned variables only. Moreover singleton consistency can be used as a preprocessing technique only or during the search but only near the root of the search tree as in [19]. We study the impact in terms of node and time efficiency of these different restriction strategies in Section 5.

3.3 Theoretical results

Following the definition in [8], we say that a local consistency Lc_1 is *stronger* than another local consistency Lc_2 if in any WCSP in which Lc_1 holds, Lc_2 holds too. By definition of singleton consistency, singleton Lc cannot be weaker than Lc. It can be strictly stronger as it was observed on random and structured binary WCSPs (see Section 5). If Lc_1 is stronger than Lc_2 then singleton Lc_1 is stronger than singleton Lc_2. The proof is given in [19] (see Theorem 2). Figure 2 shows the relation between the different local consistencies including NC*, AC*, DAC*, FDAC*, and their combinations with singleton consistency.

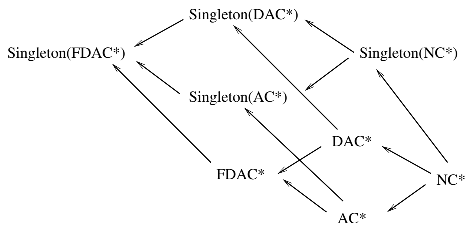

Fig. 2. Relation between local consistencies. An arrow $Lc_1 \leftarrow Lc_2$ means Lc_1 is strictly stronger than Lc_2. A missing arrow between Lc_1 and Lc_2 means both consistencies are incomparable w.r.t the stronger relation.

4 Dominance rules

Another way to simplify WCSPs is to use dominance rules. In this section, we study two dominance rules defined in [12] which allows to restrict the search space by removing values of a WCSP without changing its optimum. There is a relation between dominance and substitutability. Substitutability in CSPs has been introduced by Freuder [10]. A value (i, a) is *fully substitutable* for a value (i, b) if substituting (i, a) in any solution involving (i, b) yields another solution. We can remove b from the problem, knowing that we have not removed all the solutions. Computing full substitutability is intractable because it requires to find all solutions. The notion of *neighborhood substitutability* considers only the constraints involving a given variable. A value (i, a) is *neighborhood substitutable* for a value (i, b) if for every constraint involving the variable i, for every tuple that admits (i, b) there is an otherwise identical tuple that admits (i, a). Of course, neighborhood substitutability always implies full substitutability. Finding all neighborhood substitutable values in a CSP can be computed in polynomial time [10]. A direct generalization of substitutability to soft CSP is given by Bistarelli *et al* [3]. A value (i, a) is fully substitutable for a value (i, b) in a soft CSP if for any solution with the assignment (i, b), the identical solution, except that a is assigned to i, has a better quality. A value (i, a) is neighborhood substitutable for a value (i, b) in a soft CSP if for any tuple involving all the variables in the neighborhood of i with the assignment (i, b), the identical tuple, except that a is assigned to i, has a better quality when only considering the constraints on i. Because only optimal solutions are of interest, Bistarelli *et al* define the relaxed notion of ${}_{\alpha}$substitutability [3]. A value (i, a) is fully substitutable for a value (i, b) in a soft CSP if for any solution with the assignment (i, b) the quality of which is better than α, the identical solution, except that a is assigned to i, has a better quality. The same relaxation holds for soft neighborhood substitutability. Dominance corresponds to ${}_{\alpha}$substitutability with α equal to the optimum. A value (i, a) dominates a value (i, b) if substituting (i, a) in any optimal solution involving (i, b) yields another optimal solution. We can remove b from the problem, knowing that we have not removed all the optimal solutions. Unfortunately, in the case of non-idempotent cost aggregation operators as in WCSP, finding all neighborhood ${}_{\alpha}$substitutable values is exponential in the size of the neighborhood and quadratic in the size of the domain [3].

Koster has proposed in [12] two tractable local notions of dominance, weaker than neighborhood ${}_{\alpha}$substitutability, for the RLFAP problem [5] which is a binary WCSP with unary and binary weighted constraints. The first rule consists in computing for any variable i, an upper bound $ub(i)$ of the subproblem $\mathcal{P}^{neighborhood(i)}$ composed of variable i and its neighbors, unary cost C_i, and binary costs $C_{i,j}$.

$$ub(i) = \min_{a \in D_i}(C_i(a) + \sum_{C_{i,j} \in \mathcal{C}}(\max_{u \in D_j}(C_{i,j}(a, u))))$$

For any assignment of the neighbors of i, there exists a value in D_i such that the cost of C_i plus the costs of $C_{i,j}$ is lower than or equal to $ub(i)$. Thus, any value b in D_i having $C_i(b) > ub(i)$ can be removed because it is dominated by $argmin_{a \in D_i}(C_i(a) + \sum_{C_{i,j} \in \mathcal{C}}(\max_{u \in D_j}(C_{i,j}(a, u))))$. The time complexity of verifying this rule on all variables is $O(n^2 d^2)$. Moreover, if $ub(i) = 0$ then variable i and its constraints can be eliminated from the problem[3].

[3] In [12], a stronger rule is proposed by computing an upper bound $ub(S)$ of a set of variables. Because of its high time complexity, a unique assignment of S is heuristically chosen instead of minimizing ub for all the assignments of S.

The second rule, we called Δ_rule in the experiments, consists in computing for any variable i and for any pair of different values $a, b \in D_i \times D_i$, the maximum difference in cost of $\mathcal{P}^{neighborhood(i)}_{|D_i=\{a\}}$ and $\mathcal{P}^{neighborhood(i)}_{|D_i=\{b\}}$.

$$\Delta(i, a, b) = C_i(a) - C_i(b) + \sum_{C_{i,j} \in \mathcal{C}} (\max_{u \in D_j}(C_{i,j}(a, u) - C_{i,j}(b, u))))$$

If $\Delta(i, a, b) \leq 0$ then for any assignment of the neighbors of i, the cost of C_i plus the costs of $C_{i,j}$ is better (or the same) with assignment (i, a) than with (i, b). Value (i, a) dominates value (i, b) and b can be removed. The time complexity of verifying Δ_rule on all variables is $O(n^2d^3)$. Notice that any value removed by the first rule is also removed by the second rule, because $\sum_{C_{i,j} \in \mathcal{C}}(\max_{u \in D_j}(C_{i,j}(a, u)))) >= \sum_{C_{i,j} \in \mathcal{C}}(\max_{u \in D_j}(C_{i,j}(a, u) - C_{i,j}(b, u))))$. In the sequel, we consider only the second rule. When a value is removed, local consistency such as FDAC* is reenforced and may change the cost functions. Thus, we can iterate Δ_rule and Lc until a fix point is reached (called Δ_rule^{iter}(Lc) in the experiments).

5 Experiments

In this section we report the results of an empirical evaluation of singleton consistency and dominance on a set of benchmarks. The benchmarks[4] are binary WCSPs composed of:

- Random binary Max-CSP instances created by *random_vcsp* generator[5] using the well-known four-parameter model [24]. A random Max-CSP class is defined by $\langle n, d, e, t\rangle$ where n is the number of variables, d is the domain size, e is the number of binary constraints (i.e, graph *connectivity* given as a percentage w.r.t. a complete graph), and t the number of forbidden tuples in each constraint (i.e, *tightness* given as a percentage w.r.t. a completely forbidden constraint). Pairs of constrained variables and their forbidden tuples are randomly selected using a uniform distribution. Samples have 50 instances (only 10 instances for the *lSL* category) and we report average values. Five different categories of problems were generated following the protocol described in [14]:
  - *sparse loose (SL)* ($n = 40, d = 10, e = 13\%, t = 60\%$),
  - *large sparse loose (lSL)* ($(n, e) \in \{(50, 10), (60, 8), (70, 7), (80, 6)\}, d = 10, t = 60\%$),
  - *sparse tight (ST)* ($n = 25, d = 10, e = 21\%, t = 85\%$),
  - *dense loose (DL)* ($n = 30, d = 10, e = 25\%, t = 48\%$),
  - *dense tight (DT)* ($n = 25, d = 10, e = 25\%, t = 87\%$),
- Radio Link Frequency Assignment Problem (RLFAP). It is a communication problem where the goal is to assign frequencies to a set of radio links in such a way that all the links may operate together without noticeable interference [5]. Every

[4] Available at the *Benchmarks* link of the *SoftCSP* web site in `wcsp` format `http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/SoftCSP`.

[5] http://www.inra.fr/bia/ftp/T/VCSP

constraint is an absolute distance between a pair of frequencies (f_i, f_j) that should be greater than a constant d_{ij} ($|f_i - f_j| > d_{ij}$). Some RLFAP instances can be naturally cast as binary WCSPs where each forbidden tuple has an associated penalty cost.

We always use the local consistency FDAC* (inherited in case of singleton consistency). Our implementation is based on a free WCSP solver called TOOLBAR (C code)[6]. For all instances, we provided a good initial upper bound (optimal in the case of Max-CSPs, except for the *lSL* category where the upper bounds were produced by INCOP[16]). The experiments were performed on a 2,4 GHz Xeon computer with a time limit set to 600 seconds, except for the results in Table 2 obtained on a 2 GHz Pentium IV computer. We used default configuration parameters for TOOLBAR, except for the variable ordering heuristic. We tried on the random instances different variable ordering heuristics available in TOOLBAR. Each heuristic performed the best on different categories. In the sequel, we chose the simple static heuristic *max degree* which is independent of the pruning efficiency of our different algorithms.

The results in Table 1 show the ratio in terms of nodes and in terms of CPU time of our singleton algorithms compared to the state-of-the-art algorithm MFDAC* [14]. The speed up in terms of nodes was up to two orders of magnitude. But the CPU time was always worse (up to 3 times slower) when using singleton consistency. It is not a surprise as it was also observed on random binary/nary CSPs in [19]. Using constructive disjunction (Singleton$^{iter}_{cdisj}$(FDAC*)) did not improve upon classical singleton consistency. Using subproblem aggregation (Singleton$^{iter}_{lb}$(FDAC*)) made better results than Singleton^{iter}(FDAC*) in terms of node and time ratios, especially when the constraints are tight because the lower bound of every singleton test is often greater than the current lower bound, allowing subproblem aggregation. Notice that Singleton(FDAC*) without the **repeat** loop still offered large saving in the number of nodes and was up to 33% faster in CPU time than Singleton^{iter}(FDAC*).

	SL	ST	DL	DT
Singleton(FDAC*)	55.3 n., 1.8 s.	26.4 n., 2.4 s.	41.1 n., 2.3 s.	26.1 n., 2.5 s.
Singleton^{iter}(FDAC*)	78.0 n., 2.4 s.	34.5 n., 2.9 s.	60.7 n., 3.1 s.	34.7 n., 2.9 s.
Singleton$^{iter}_{cdisj}$(FDAC*)	78.1 n., 2.4 s.	34.6 n., 2.8 s.	60.8 n., 3.1 s.	34.8 n., 3.0 s.
Singleton$^{iter}_{lb}$(FDAC*)	100.3 n., 2.0 s.	76.2 n., 1.8 s.	64.8 n., 2.9 s.	107.0 n., 1.5 s.

Table 1. Performance of singleton consistencies w.r.t. MFDAC* on random binary Max-CSPs using *max degree* variable ordering heuristic. *n.*: inverse ratio MFDAC*/Singleton in terms of nodes; *s.*: ratio Singleton/MFDAC* in terms of CPU time.

When further restricting the singleton consistency algorithm by performing singleton tests on the most constrained variables only (*cf.* Section 3.2), then singleton consistency obtained better results in CPU time than MFDAC* on some random binary

[6] TOOLBAR including singleton consistency and dominance is available at the *Algorithms* link of the *SoftCSP* web site.

Max-CSPs. We tried 10%, 25%, and 100% of selected variables for singleton tests. Figure 3 shows the results on large sparse loose problems (*lSL* category). The time limit was set to one hour. By default, singleton consistency follows the lexicographic order when it examines the variables. When following the *max degree* order, even Singleton(100,MFDAC*) got slightly better results than MFDAC*. The best results as reported in Figure 3 were obtained with 25% of selected variables. We observed that, in most cases, values which are removed by constructive disjunction are also removed by classic singleton consistency. But $\mathsf{Singleton}_{cdisj}$(FDAC*) removed values earlier w.r.t. the variable order used by singleton consistency compared to Singleton(FDAC*). If singleton consistency is applied to a subset of the variables only, then constructive disjunction can improve the results. The combination $\mathsf{Singleton}_{cdisj+lb}$(25,FDAC*) got the best results. The relative gain (100*(MFDAC* - $\mathsf{Singleton}_{cdisj+lb}$(25,FDAC*))/MFDAC*) in CPU time went from 27% to 41% for different problem sizes of the *lSL* category. This gain globally increased as the problem size increases.

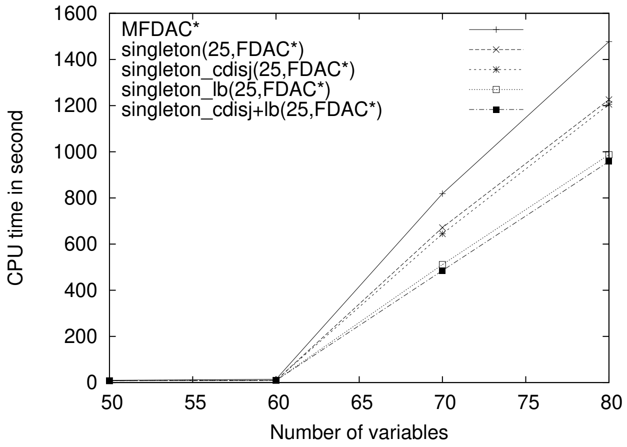

Fig. 3. Comparison between MFDAC* and restricted singleton consistencies on large sparse loose random binary Max-CSPs (*lSL* category) using *max degree* variable ordering heuristic.

We applied singleton consistency and the dominance rule $\Delta\_\mathsf{rule}^{iter}$ as a preprocessing technique on the RLFAP benchmark. Table 2 shows the percentage of removed values using dominance and singleton consistency. Both methods could prune more than FDAC* and their combination was always better, showing that dominance and singleton consistency are complementary techniques. Notice the very good performance of FDAC* which proved optimality of five instances without backtracking. Singleton consistency proved optimality on one more instance (scen09). On graph11, 50% of the values were removed by using dominance and singleton consistency in 166 seconds. This good pruning behavior also had the effect of improving the lower bound of graph11: FDAC* deduced a lower bound equal to 2558, the lower bound found by our improved singleton consistency was equal to 2683, and the best known upper bound is equal to 3080.

$\mathcal{P}$	graph05	graph06	graph07	graph11	graph12	graph13	scen06	scen07	scen08	scen09	scen10
$\lvert\mathcal{D}\rvert$	3708	7542	5160	12820	9502	17588	4010	7976	18100	7876	7876
FDAC*	100.0	100.0	100.0	16.1	100.0	0.9	0.1	9.6	0.0	69.7	100.0
Δ	100.0	100.0	100.0	17.6	100.0	1.4	7.5	13.4	4.8	91.7	100.0
singl.	100.0	100.0	100.0	45.7	100.0	9.5	0.8	20.7	0.0	100.0	100.0
All	100.0	100.0	100.0	50.3	100.0	9.9	8.2	26.6	4.8	100.0	100.0

Table 2. Dominance rule and singleton consistency used as a preprocessing technique on the RLFAP benchmark. $|\mathcal{D}|$ is the total initial number of values in each instance. Other numbers are the percentage of removed values w.r.t. $|\mathcal{D}|$. Δ: Δ_rule^{iter}(FDAC*), *singl.*: Singleton$^{iter}_{cdisj+lb}$(FDAC*), *All*: Δ_rule^{iter}(Singleton$^{iter}_{cdisj+lb}$(FDAC*)).

6 Discussion and conclusion

In the CSP framework, there were several attempts to reduce the average or the worst-case time complexity of singleton arc consistency algorithms. We briefly present these approaches and see if they can be adapted to the WCSP framework. As pointed out by [19], we are not interested in what values need to be pruned to make the subproblem $\mathcal{P}_{|D_i=\{a\}}$ in line 3 locally consistent. Following the lazy AC7 algorithm [23], we could derive a lazy FDAC* algorithm which looks for a sub-domain $D' = (D'_1, \ldots, D'_n)$ such that $\forall i, \emptyset \subset D'_i \subseteq D_i$ and all the values in D' have a support or a full support in D'. The lazy approach can also be applied to the Singleton() algorithm by ignoring value removals in $\mathcal{P}$ which do not increase the lower bound. This can be done by checking the supports of FDAC* in $\mathcal{P}$. If the value (i, a) is not the support of any variable (NC* supports) nor any value (AC* and DAC* supports) in $\mathcal{P}$ then we do not need to enforce FDAC* on $\mathcal{P}_{|D_i=\{a\}}$ in line 3. Testing for this particular case can be done in constant time by updating a boolean array as soon as a new (full) support is found in FDAC*. Using TOOLBAR, we analyzed how many values are not a support just after FDAC* is applied to the root node. We found from 30% up to 60% of non-support values in our benchmark of random loose Max-CSPs. This number should increase by using lazy FDAC*.

In [1], the average time complexity of singleton arc consistency is improved. "Like in AC-4, the idea is to keep a set of values supported by a given value. All the values in the domains of $\mathcal{P}_{|D_i=\{a\}}$ after making this problem arc consistent are supporters for singleton arc consistency of (i, a)" [1]. When a value is removed, the singleton test is performed on the set of supported values only. Experiment results in the phase transition area of random binary CSPs show a 40% saving in singleton tests. The same approach can be directly applied to WCSPs (forbidden values in $\mathcal{P}_{|D_i=\{a\}}$ are detected if an initial upper bound is provided). Notice that when using a restricted singleton consistency without the **repeat** loop, the gain in singleton tests may be rather small during the search because at each node of the search tree, all the values except one are removed when a variable is assigned thus almost all the singleton tests have to be performed again.

An optimal singleton arc consistency algorithm for CSPs is proposed in [2] the worst-case time complexity of which is $O(end^3)$ instead of $O(en^2d^4)$. By duplicating the problem nd times, one subproblem $\mathcal{P}_{|D_i=\{a\}}$ for each value (i,a), this algorithm can benefit from the incrementality of arc consistency on each of them. When a value is found singleton inconsistent, the **repeat** loop performs all the singleton tests again, one for each subproblem, without performing arc consistency from scratch. The question is whether the additional overhead of duplications during the search pays-off.

In conclusion, we have proposed two improvements on a basic singleton consistency algorithm. The first one is based on constructive disjunction. It can detect with a small overhead forbidden values, which has the effect to diminish the number of singleton tests to be performed. It is best combined with restricted singleton consistency which examines the most constrained variables only. Such a technique can be applied on WCSPs and CSPs. We showed that in the case of random binary Max-CSPs, it can perform better than a state-of-the-art WCSP solver. The second improvement consists in an opportunistic dynamic variable ordering for directional arc consistency (DAC*). When the lower bound is improved by node consistency in each singleton test for a given variable i, it enforces directional arc consistency on this variable with all the constraints oriented towards i. The result is an equivalent problem having a better lower bound. We also studied two dominance rules defined in [12] and showed their relations between themselves and with soft substitutability [3]. The conjunction of singleton consistency and dominance gave interesting results on the RLFAP benchmark [5]. Notice that our improved singleton consistency algorithm is still valid in the case of n-ary constraints. The dominance rules are restricted to binary WCSPs.

Further studies should address the design of better variable ordering heuristics to branch in the search tree, based on the information provided by the singleton tests, as it has been done for satisfiability problems [15]. Another research direction is to develop an efficient algorithm for maintaining the Δ_rule dominance rule during the search.

Acknowledgements We would like to thank Thomas Schiex for his fruitful comments on the paper.

References

[1] BARTÁK, R., AND ERBEN, R. A new algorithm for singleton arc consistency. In *Proc. of FLAIRS-2004* (2004), AAAI Press.
[2] BESSIÈRE, C. An optimal algorithm for sac. Coconut Note 77, 2004.
[3] BISTARELLI, S., FALTINGS, B., AND NEAGU, N. Interchangeability in soft csps. In *Recent Advances in Constraints, Joint ERCIM/CologNet International Workshop on Constraint Solving and Constraint Logic Programming. Selected Papers.* (2003), vol. 2627, Springer Lecture Notes in Artificial Intelligence.
[4] BISTARELLI, S., MONTANARI, U., AND ROSSI, F. Semiring based constraint solving and optimization. *Journal of the ACM 44*, 2 (1997), 201–236.
[5] CABON, B., DE GIVRY, S., LOBJOIS, L., SCHIEX, T., AND WARNERS, J. Radio link frequency assignment. *Constraints Journal 4* (1999), 79–89.

[6] COOPER, M. Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets and Systems 134*, 3 (2003), 311 – 342.

[7] DE GIVRY, S., LARROSA, J., MESEGUER, P., AND SCHIEX, T. Solving max-sat as weighted csp. In *Proc. of CP-03* (Kinsale, County Cork, Ireland, 2003), pp. 363–376.

[8] DEBRUYNE, R., AND BESSIÈRE, C. Some practicable filtering techniques for the constraint satisfaction problem. In *Proc. of the 15th IJCAI* (Nagoya, Aichi, Japan, 1997), pp. 412–417.

[9] FREUDER, E. C. Partial constraint satisfaction. In *Proc. of the 11th IJCAI* (Detroit, MI, 1989), pp. 278–283.

[10] FREUDER, E. C. Eliminating interchangeable values in constraint satisfaction problems. In *Proc. of AAAI'91* (Anaheim, CA, 1991), pp. 227–233.

[11] JOURDAN, J., AND SOLA, T. The versatility of handling disjunctions as constraints. In *PLILP 1993* (Tallinn, Estonia, 1993), pp. 60–74.

[12] KOSTER, A. M. *Frequency assignment: Models and Algorithms*. PhD thesis, University of Maastricht, The Netherlands, Nov. 1999. Available at www.zib.de/koster/thesis.html.

[13] LARROSA, J. On arc and node consistency in weighted CSP. In *Proc. AAAI'02* (Edmondton, (CA), 2002), pp. 48–53.

[14] LARROSA, J., AND SCHIEX, T. In the quest of the best form of local consistency for weighted CSP. In *Proc. of the 18th IJCAI* (Acapulco, Mexico, Aug. 2003), pp. 239–244.

[15] LI, C. M., AND ANBULAGAN. Heuristics based on unit propagation for satisfiability problems. In *Proc. of the 15th IJCAI* (Nagoya, Aichi, Japan, 1997), pp. 366–371.

[16] NEVEU, B., AND TROMBETTONI, G. INCOP: An Open Library for INcomplete Combinatorial OPtimization. In *Proc. of CP-03* (Cork, Ireland, 2003), pp. 909–913.

[17] PETIT, T. *Modélisation et Algorithmes de Résolution de Problèmes Sur-Contraints*. PhD thesis, Université Montpellier II, 2002.

[18] PETIT, T., REGIN, J.-C., AND BESSIÈRE, C. Generalization of constructive disjunction for over-constrained problems. In *INFORMS'01* (2001).

[19] PROSSER, P., STERGIOU, K., AND WALSH, T. Singleton consistencies. In *Proc. of CP-00* (Singapore, 2000), pp. 353–368.

[20] SCHIEX, T. Arc consistency for soft constraints. In *Principles and Practice of Constraint Programming - CP 2000* (Singapore, Sept. 2000), vol. 1894 of *LNCS*, pp. 411–424.

[21] SCHIEX, T. Bayesian networks and Soft constraints. Working paper, 2002.

[22] SCHIEX, T., FARGIER, H., AND VERFAILLIE, G. Valued constraint satisfaction problems: Hard and easy problems. In *Proc. of IJCAI-95* (Montréal, Canada, 1995), pp. 631–637.

[23] SCHIEX, T., RÉGIN, J.-C., GASPIN, C., AND VERFAILLIE, G. Lazy arc consistency. In *Proc. of AAAI-96* (Portland, OR, 1996), pp. 216–221.

[24] SMITH, B. The phase transition in constraint satisfaction problems: A closer look at the mushy region. In *Proc. of the 11st ECAI* (Amsterdam, The Netherlands, 1994).

[25] VAN HENTENRYCK, P. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, The MIT Press, 1989.

[26] WÜRTZ, J., AND MÜLLER, T. Constructive disjunction revisited. In *20th German Annual Conference on Artificial Intelligence* (Dresden, Germany, 1996), G. Görz and S. Hölldobler, Eds., vol. 1137, Springer-Verlag, pp. 377–386.