Super Solutions in Constraint Programming

Emmanuel Hebrard, Brahim Hnich, and Toby Walsh*

Cork Constraint Computation Centre
University College Cork
{e.hebrard, brahim, tw}@4c.ucc.ie

Abstract. To improve solution robustness, we introduce the concept of super solutions to constraint programming. An (a, b)-super solution is one in which if a variables lose their values, the solution can be repaired by assigning these variables with a new values and at most b other variables. Super solutions are a generalization of supermodels in propositional satisfiability. We focus in this paper on (1,0)-super solutions, where if one variable loses its value, we can find another solution by re-assigning this variable with a new value. To find super solutions, we explore methods based both on reformulation and on search. Our reformulation methods transform the constraint satisfaction problem so that the only solutions are super solutions. Our search methods are based on a notion of super consistency. Experiments show that super MAC, a novel search-based method shows considerable promise. When super solutions do not exist, we show how to find the most robust solution. Finally, we extend our approach from robust solutions of constraint satisfaction problems to constraint optimization problems.

1 Introduction

Where changes to a solution introduce additional expenses or reorganization, solution robustness is a valuable property. A robust solution is not sensitive to small changes. For example, a robust schedule will not collapse immediately when one job takes slightly longer to execute than planned. The schedule should change locally and in small proportions, and the overall makespan should change little if at all. To improve solution robustness, we introduce the concept of super solutions to constraint programming (CP). An (a, b)-super solution is one in which if the values assigned to a variables are no longer available, the solution can be repaired by assigning these variables with a new values and at most b other variables. An (a, b)-super solution is a generalization of both fault tolerant solutions in CP [18] and supermodels in propositional satisfiability (SAT) [12]. We show that finding (a, b)-super solutions for any fixed a is NP-Complete in general. Super solutions are computed offline and do not require knowledge about the likely changes. A super solution guarantees the existence of a small set of repairs when the future changes in a small way.

* All authors are supported by the Science Foundation Ireland and an ILOG grant.

In this paper, we focus on the algorithmic aspects of finding (1,0)-super solutions, which are the same as fault tolerant solutions [18]. A (1,0)-super solution is a solution where if one variable loses its value, we can find another solution by re-assigning this variable with a new value, and no other changes are required for the other variables. We explore methods based both on reformulation and on search to find (1,0)-super solutions. Our reformulation methods transform the constraint satisfaction problem so that the only solutions are super solutions. We review two reformulation techniques presented in [18], and introduce a new one, which we call the cross-domain reformulation. Our search methods are based on notions of super consistency. We propose two new search algorithms that extend the maintaining arc consistency algorithm (MAC [9, 8]). We empirically compare the different methods and observe that one of them, super MAC shows considerable promise. When super solutions do not exist, we show how to find the most robust solution closest to a super solution. We propose a super Branch & Bound algorithm that finds the most robust solution, i.e., a solution with the maximum number of repairable variables. Finally, we extend our approach from robust solutions of constraint satisfaction problems to constraint optimization problems. We show how an optimization problem becomes a multi-criterion optimization problem, where we optimize the number of repairable variables and the objective function.

2 Super Solutions

Supermodels were introduced in [12] as a way to measure solution robustness. An (a, b)-supermodel of a SAT problem is a model (a satisfying assignment) with the additional property that if we modify the values taken by the variables in a set of size at most a (breakage variables), another model can be obtained by flipping the values of the variables in a disjoint set of size at most b (repair variables).

There are a number of ways we could generalize the definition of supermodels from SAT to CP as variables now can have more than two values. A break could be either "losing" the current assignment for a variable and then freely choosing an alternative value, or replacing the current assignment with some other value. Since the latter is stronger and potentially less useful, we propose the following definition.

Definition 1. *A solution to a CSP is (a, b)-super solution iff the loss of the values of at most a variables can be repaired by assigning other values to these variables, and modifying the assignment of at most b other variables.*

Example 1. Let us consider the following CSP: $X, Y, Z \in \{1, 2, 3\} \quad X \leq Y \wedge Y \leq Z$. The solutions to this CSP are shown in Figure 1, along with the subsets of the solutions that are $(1, 1)$-super solutions and $(1, 0)$-super solutions.

The solution $\langle 1, 1, 1\rangle$ is not a $(1, 0)$-super solution. If X loses the value 1, we cannot find a repair value for X that is consistent with Y and Z since

solutions	$(1,1)$-super solutions	$(1,0)$-super solutions
$\langle 1,1,1\rangle$, $\langle 1,1,2\rangle$	$\langle 1,1,2\rangle$, $\langle 1,1,3\rangle$	$\langle 1,2,3\rangle$
$\langle 1,1,3\rangle$, $\langle 1,2,2\rangle$	$\langle 1,2,2\rangle$, $\langle 1,2,3\rangle$	$\langle 1,2,2\rangle$
$\langle 1,2,3\rangle$, $\langle 1,3,3\rangle$	$\langle 1,3,3\rangle$, $\langle 2,2,2\rangle$	$\langle 2,2,3\rangle$
$\langle 2,2,2\rangle$, $\langle 2,2,3\rangle$	$\langle 2,2,3\rangle$, $\langle 2,3,3\rangle$	
$\langle 2,3,3\rangle$, $\langle 3,3,3\rangle$		

Fig. 1. solutions, $(1,1)$-super solutions, and $(1,0)$-super solutions for the CSP: $X \leq Y \leq Z$.

neither $\langle 2,1,1\rangle$ nor $\langle 3,1,1\rangle$ are solutions. Also, solution $\langle 1,1,1\rangle$ is not a $(1,1)$-super solution since when X loses the value 1, we cannot repair it by changing the value assigned to at most one other variable, i.e., there exists no repair solution when X breaks since none of $\langle 2,1,1\rangle$, $\langle 3,1,1\rangle$, $\langle 2,2,1\rangle$, $\langle 2,3,1\rangle$, $\langle 2,1,2\rangle$, and $\langle 2,1,3\rangle$ is a solution. On the other hand, $\langle 1,2,3\rangle$ is a $(1,0)$-super solution since when X breaks we have the repair solution $\langle 2,2,3\rangle$, when Y breaks we have the repair solution $\langle 1,1,3\rangle$, and when Z breaks we have the repair solution $\langle 1,2,2\rangle$. We therefore have a theoretical basis to prefer the solution $\langle 1,2,3\rangle$ to $\langle 1,1,1\rangle$, as the former is more robust.

A number of properties follow immediately from the definition. For example, a (c,d)-super solution is a (a,b)-super solution if ($a \leq c$ or $d \leq b$) and $c+d \leq a+b$. Deciding if a SAT problem has an (a,b)-supermodel is NP-complete [12]. It is not difficult to show that deciding if a CSP has an (a,b)-super solution is also NP-complete, even when restricted to binary constraints.

Theorem 1. *Deciding if a CSP has an* (a,b)*-super solution is NP-complete for any fixed* a*.*

Proof. To see it is in NP, we need a polynomial witness that can be checked in polynomial time. This is simply an assignment which satisfies the constraints, and, for each of the $O(n^a)$ (which is polynomial for fixed a) possible breaks, the $a+b$ repair values.

To show completeness, we show how to map a binary CSP onto a new binary problem in which the original has a solution iff the new problem has an (a,b)-super solution. Our reduction constructs a CSP which, if it has any solution, has an (a,b)-super solution for any $a+b \leq n$. The problem will even have a $(n,0)$-super solution. It is possible to show that if we have a $(n,0)$-super solution then we also have an (a,b)-super solution for any $a+b \leq n$. However, we will argue *directly* that the original CSP has a solution iff the constructed problem has an (a,b)-super solution.

We duplicate the domains of each of the variables, and extend the constraints so that the behave equivalently on the new values. For example, suppose we have a constraint $C(X,Y)$ which is only satisfied by $C(m,n)$. Then we extend the constraint so that is satisfied by just $C(m,n)$, $C(m',n)$, $C(m,n')$ and $C(m',n')$ where m' and n' are the duplicated values for m and n. Clearly, this binary CSP has a solution iff the original problem also has. In addition, any break of

a variables can be repaired by replacing the a corresponding values with their primed values (or unpriming them if they are already primed) as well as any b other values. □

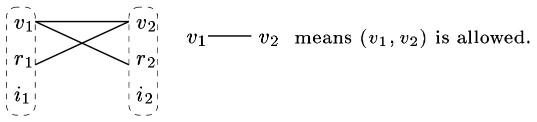
A necessary but not sufficient condition to find supermodels in SAT or super solutions in CSPs is the absence of backbone variables. A *backbone variable* is a variable that takes the same value in all solutions. As a backbone variable has no alternative, a SAT or CSP problem with a backbone variable cannot have any (a, b)-supermodels or (a, b)-super solutions.

Another important factor that influences the existence of super solutions is the way the problem is modeled. For instance, the direct encoding into SAT (i.e., one Boolean variable for each pair variable-value in the CSP [15]) of the problem in Example 1 has no $(1, 0)$-supermodels, even though the original CSP had a $(1, 0)$-super solution. Moreover, the meaning of a super solution depends on the model. For example, if a variable is a job and a value is a machine, the loss of a value may mean that the machine has now broken. On the other hand, if a variable is a machine and the value is a job, the loss of a value may mean that the job is now not ready to start. The CP framework gives us more freedom than SAT to choose what variables and values stand for, and therefore to the meaning of a super solution. For the rest of the paper, we just focus on (1,0)-super solutions and refer to them as super solutions when it is convenient.

3 Finding $(1, 0)$-Super Solutions Via Reformulation

Fault tolerant solutions [18] are the same as (1,0)-super solutions. The first reformulation approach in [18] allows only fault tolerant solutions, but not *all* of them (see [3] for a counter example). The second approach in [18] duplicates the variables. The duplicate variables have the same domain as the original variables, and are linked by the same constraints. A not equals constraint is also posted between each original variable and its duplicate. The assignment to the original variables is a super solution, where the repair for each variable is given by its duplicate. We refer to the reformulation of a CSP P with this encoding as P+P.

We now present a third and new reformulation approach. Let $S = \langle v_1, v_2 \rangle$ be part of a $(1, 0)$-super solution on *two* variables X and Y. If v_1 is lost, then there must be a value $r_1 \in \mathcal{D}(X)$ that can repair v_1, that is $\langle r_1, v_2 \rangle$ is a compatible tuple. Symmetrically, there must exists r_2 such that $\langle v_1, r_2 \rangle$ is allowed. Now consider the following subproblem involving two variables:



Since it satisfies the criteria above, $S = \langle v_1, v_2 \rangle$ is a super solution whilst any other tuple is not. One may try to prune the values r_1, r_2, i_1,and i_2 as they do not participate in any super solution. However r_1 and r_2 are *essential* for *providing* support to v_1 and v_2. On the other hand, i_1 and i_2 are simply not supported and can thus be pruned. So, we cannot simply reason about extending partial instantiations of values, unless we keep the information about the values that can be used as repair. So, let us instead think of the domain of the variables as pairs of values $\langle v, r\rangle$, the first element corresponding to the *super value* (which is part of a super solution), the second corresponding to the *repair value* (which can repair the former). Our cross-domain reformulation exploits this. We reformulate a CSP $P = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$ such that any domain becomes its own cross-product (less the doubletons),i.e. $\mathcal{D}(\mathcal{X})$ becomes $\mathcal{D}(\mathcal{X}) \times \mathcal{D}(\mathcal{X}) - \{\langle v, v\rangle | v \in \mathcal{D}(\mathcal{X})\}$. The constraints are built as follows. Two pairs $\langle v_1, r_1\rangle$ and $\langle v_2, r_2\rangle$ are compatible iff

- v_1 and v_2 are compatible (the solution must be consistent at the first place);
- v_1 and r_2 are compatible (in case of a break involving v_2, r_2 can be a repair);
- v_2 and r_1 are compatible (in case of a break involving v_1, r_1 can be a repair).

The new domain $\mathcal{CD}(\mathcal{X})$ and $\mathcal{CD}(\mathcal{Y})$ of variable X and Y are:

$$\{\langle v_1, r_1\rangle, \langle v_1, i_1\rangle, \langle r_1, v_1\rangle, \langle r_1, i_1\rangle, \langle i_1, v_1\rangle, \langle i_1, r_1\rangle\}$$

$$\{\langle v_2, r_2\rangle, \langle v_2, i_2\rangle, \langle r_2, v_2\rangle, \langle r_2, i_2\rangle, \langle i_2, v_2\rangle, \langle i_2, r_2\rangle\}$$

The only one allowed tuple is $S = \langle\langle v_1, r_1\rangle, \langle v_2, r_2\rangle\rangle$. We refer to the cross-domain reformulation of a problem P as $P \times P$.

4 Finding $(1, 0)$-Super Solutions Via Search

We first introduce the notion of super consistency for binary constraints, and then use it to build some new search algorithms.

4.1 Super Consistency

Backtrack-based search algorithms like MAC use local consistency to detect unsatisfiable subproblems. Local consistency can also be used to develop efficient algorithms for finding super solutions. We shall introduce three ways of incorporating arc consistency (AC) into a search algorithm for seeking super solutions.

AC+ is a naive approach that augments the traditional AC by a further condition, achieving a very low level of filtering.

AC($P \times P$) maintains AC on the cross-domain reformulation of P. This method allows us to infer *all* that can be inferred locally, just as AC does in a regular CSP [5]. However, this comes at a high polynomial cost.

Super AC gives less inference than AC($P \times P$), but is a good tradeoff between the amount of pruning and complexity.

Informally, the consistent closure of a CSP contains only partial solutions for a given level of locality. However, the situation with super solutions is more complex because values that do not get used in any local super solution can still be essential as a *repair* and thus cannot be simply pruned.

AC+: If S is a super solution, then for every variable, at least *two* values are consistent with all the others values of S. Consequently, being arc consistent *and having non-singleton domains* is a necessary condition for the existence of super solution. AC+ is therefore defined as follows: for a CSP $P = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$: AC+($P$) $\Leftrightarrow$AC(P) $\wedge \forall D \in \mathcal{D}, |D| > 1$. Whilst AC+ is usually too weak to give good results, it is the basis for an algorithm for the associated optimization problem (discussed in section 5).

AC($P\times P$): AC on $P\times P$ is the tightest local domain filtering possible. Note that $P\times P$ also has the same constraint graph topology as the original problem P. As a corollary, if the constraint graph of P is a tree, we can use AC on $P\times P$ to find $(1, 0)$-super solutions in polynomial time.

Super AC: AC on $P\times P$ allows us to infer all that can be inferred locally. In other words, we will prune any value in a cross-domain that is not locally consistent. However, this comes at high cost. Maintaining AC will be $O(d^4)$ where d is the initial domain size. We therefore propose an alternative that does less inference, but at just $O(d^2)$ cost.

The main reason for the high cost is the size of the cross-domains. A cross-domain is quadratic in the size of the original domain since it explicitly represents the repair value for each super value. Here we will simulate much of the inference performed by super consistency, but will only look at one value at a time, and not pairs. We will divide the domain of the variable into two separate sets of domains:

- The "super domain" (SD) where only super values are represented;
- The "repair domain" (RD) where repair values are stored.

We propose the following definition for super AC:

- A value v is in the super domain of X iff for any other variable Y, there exists v' in super domain of Y and r in repair domain of Y such that $\langle v, v'\rangle$ and $\langle v, r\rangle$ are allowed and $v' \neq r$.
- A value v is in repair domain of X iff for any other variable Y, there exists v' in super domain of Y such that $\langle v, v'\rangle$ is allowed.

The definition of super AC translates in a straightforward way into a filtering algorithm. The values are marked as either *super* or *repair*, and when looking for support of a super value, an additional and different support marked either as *super* or *repair* is required. The complexity of checking the consistency of an arc increases only by a factor of 2 and thus remains in $O(d^2)$.

Theoretical Properties: We now show that maintaining AC on $P\times P$ achieves more pruning than maintaining super AC on P, which achieve more pruning than maintaining AC on P+P or AC+ on P (which are equivalent). For the theorem and the proof below, we use the notation (x)(P) to denote that the problem P is "consistent" for the filtering (x).

Theorem 2 (level of filtering). *For any problem P, $AC(P \times P) \Rightarrow$ super $AC(P)$ $\Rightarrow AC(P+P) \Leftrightarrow AC+(P)$.*

Proof. **(1) AC(P+P) $\Rightarrow$ AC+(P):** Suppose that P is not AC+, then in the arc consistent closure of P, there exists at least one domain D_i such that $|D_i| \leq 1$. P+P contains P. In its arc consistent closure, we have $|D_i| \leq 1$ as well. X_i is linked to a duplicate of itself in which the domain D'_i is then equal to D_i and therefore singleton (with the same value) or empty. However, recall that we force $X_i \neq X'_i$, thus P+P is not AC.

(2) AC+(P) $\Rightarrow$ AC(P+P): Suppose we have AC(P) and any domain D in P is such that $|D| > 1$, now consider P+P. The original constraints are AC since P is AC. The duplicated constraints are AC since they are identical to the original ones. The not equals constraints between original and duplicated variables are AC since any variable has at least 2 values.

(3) super AC(P) $\Rightarrow$ AC+(P): Suppose that P is not AC+, then there exists two variables X, Y such that any value of X has at most one support on Y, therefore the corresponding super domain is wiped-out, and P is not super AC.

(4) super AC(P) $\not\Leftarrow$ AC+(P): See counter-example in Figure 2.

(5) AC($P \times P$) $\Rightarrow$ super AC(P): Suppose that AC($P \times P$), then for any two variables X, Y there exist two pairs $\langle v1, r1 \rangle \in D(X) \times D(X), \langle v2, r2 \rangle \in D(Y) \times D(Y)$, such that $\langle v1, r2 \rangle, \langle r1, v2 \rangle$ and $\langle v1, v2 \rangle$ are allowed tuples. Therefore $v1$ belongs to the super domain of X and $v1$ and $r1$ belong to the repair domain of X. Thus, the super domain of X is not empty and the repair domain of X is not singleton. Therefore, P is super AC.

(6) AC($P \times P$) $\not\Leftarrow$ super AC(P): See counter-example in Figure 3. □

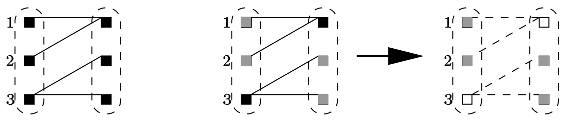


Fig. 2. The first graph shows the microstructure of a simple CSP, two variables and three values each, *allowed* combinations are linked. P is AC+ since the network is arc consistent and every domain contains 3 values. However, P is not super AC since the grayed values (in the second graph) are not in super domains, they have only one support. In the second step, the whitened variables (in the third graph) are also removed from both repair and super domains since they do not have a support in a super domain.

4.2 Super Search Algorithms

We now present two new search algorithms: MAC+ and super MAC.

MAC+ This algorithm establishes AC+ at each node. That is, it maintains AC and backtracks if a domain wipes out *or* becomes singleton. In the MAC algorithm, we only prune future variables, since the values assigned to past

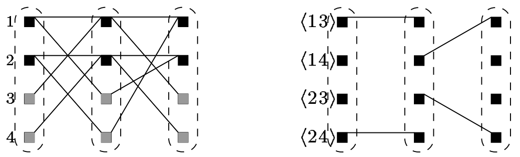


Fig. 3. The first graph shows the microstructure of a simple CSP, three variables and four values each, *allowed* combinations are linked. P is super AC since the black values have each one "black" support, and another "gray" or "black" for every neighbor, and all gray values have one "black" support. The super domains thus contain "black" values (size 2), and repair domains contain "black" and "gray" values (size 4). The second graph shows $P{\times}P$, which is not AC.

variables are guaranteed to have a support in each future variable. Here, this also holds, but the condition on the size of the domains may be violated for an assigned variable because of an assignment in the future. Therefore, AC is first established on the whole network, and not only on the future variables. Second, variables are not assigned in a regular way (e.g. by reducing their domains to the chosen value) but one value is marked as super value, that is added to the current partial solution, and unassigned values are kept in the domain as potential repairs. The algorithm can be informally described as follows:

- Choose a variable X.
- Mark a value $v \in D(X)$ as assigned, but keep the unassigned values.
- For all $Y \neq X$, backtrack if Y has less than two supports for v.
- Revise the constraints as the MAC algorithm, and backtrack if the size of any domain falls bellow 2.

Super MAC: We give the pseudo code of super MAC in Figure 4. The algorithm is very similar to the MAC algorithm. Most of the differences are grouped in the procedure `revise-Dom`. The super domains ($\mathcal{SD}$) and repair domains ($\mathcal{RD}$) are both equal to the original domains for the first call. The values are pruned by maintaining super AC (`revise-dom`, loop 1). The algorithm backtracks if a super domain wipes out or a repair domain becomes singleton (line 2). Note that, as for MAC+, super AC is also maintained on the domains of the assigned variables (`super AC`, loop 1).

We have established an ordering relation on the different filterings. However, for the two algorithms above, *assigning* a value to a variable in the current solution does not give the same subproblem as in a regular algorithm. For a regular backtrack algorithm, the domains of assigned variables are reduced to the chosen value, whilst unassigned values are still in their domain for the algorithms above. We have proved that a problem P is AC+ iff $P{+}P$ is AC. However, consider the subproblem P' induced by the *assignment* of X by MAC+. P' may have more than one value in the domain of X, whereas the corresponding assignment in $P{+}P$ leaves only one value in the domain of X (see Figure 5). Therefore the ordering on the consistencies does not lift immeditately to an ordering on the number of backtracks of the algorithms themselves. However, MAC($P{\times}P$) always backtracks when one of the other algorithms does, whilst MAC+ never

Algorithm 1: super MAC

```
Data   : CSP: $P = \{\mathcal{X}, \mathcal{SD}, \mathcal{RD}, \mathcal{C}\}$, solution: $S = \emptyset$, variables: $\mathcal{V} = \mathcal{X}$
Result : Boolean // $\exists S$ a $(1,0)$-super solution
if $\mathcal{V} = \emptyset$ then return True;
choose $X_i \in \mathcal{V}$;
foreach $v_i \in SD_i$ do
    save $\mathcal{SD}$ and $\mathcal{RD}$;
    $SD_i \leftarrow \{v_i\}$;
    if super AC($P, \{X_i\}$) then
        if super MAC($P, S \cup \{v_i\}, \mathcal{V} - \{X_i\}$) then return True;
    restore $\mathcal{SD}$ and $\mathcal{RD}$;
return False;
```

Algorithm 2: super AC

```
    Data   : CSP: $P = \{\mathcal{X}, \mathcal{SD}, \mathcal{RD}, \mathcal{C}\}$, Stack: $\{X_i\}$
    Result : Boolean // $P$ is super arc consistent
    while Stack is not empty do
        pop $X_i$ from Stack;
1       foreach $C_{ij} \in \mathcal{C}$ do
            switch revise-Dom($SD_j, RD_j, SD_i, RD_i$) do
                case not-cons
                    return False;
                case pruned
                    push $X_j$ on Stack;
    return True;
```

Procedure revise-Dom(SD_j, RD_j, SD_i, RD_i) : {pruned,not-cons,nop}

```
1   foreach $v_j \in SD_j$ do
        if $\nexists v_i \in SD_i, v'_i \in RD_i$ such that $\langle v_i, v_j\rangle \in C_{ij} \wedge \langle v'_i, v_j\rangle \in C_{ij} \wedge v_i \neq v'_i$ then
            $SD_j \leftarrow SD_j - \{v_j\}$;
    foreach $v_j \in RD_j$ do
        if $\nexists v_i \in SD_i$ such that $\langle v_i, v_j\rangle \in C_{ij}$ then
            $RD_j \leftarrow RD_j - \{v_j\}$;
    if at least one value has been pruned then return pruned;
2   if $|SD_j| = 0 \vee |RD_j| < 2$ then return not-cons;
    return nop;
```

Fig. 4. super MAC algorithm

backtracks unless all the other algorithms do. Therefore any solution found by MAC($P \times P$) will eventually be found by the others, and MAC+ will only find solutions found by one of the other algorithms. We prove that MAC+ is correct and MAC($P \times P$) is complete. Hence all four algorithms are correct and complete.

Theorem 3. *For any given CSP P, the sets of solutions of MAC+(P), of super MAC(P), of MAC(P×P), and of MAC(P+P) are identical and equal to the super solutions of P.*

Proof. MAC+ is correct: suppose that S is not a super solution, then there exists a variable X assigned to v in S, such that $\forall w \in D(X), v \neq w$, w cannot replace v in S. Therefore when all the variables are assigned, and so there remain in the

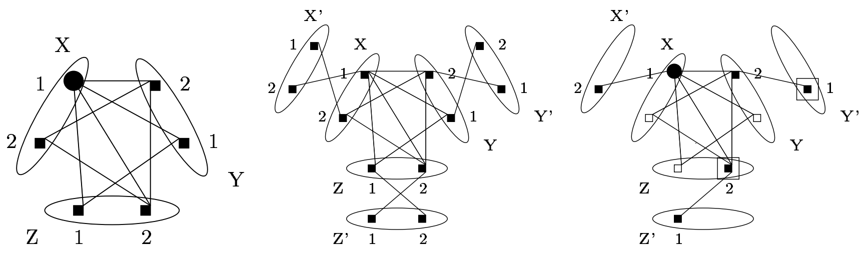


Fig. 5. Left: A CSP P, P is still AC+ after assigning X to 1. Middle: P+P, each variable has a duplicate which must be different, the constraints linking those variables are not represented here, the constraints on X' are exactly the same as the ones on X. Right: When the same assignment, $X = 1$ is done in P+P, we have the following propagation $X' \neq 1 \rightarrow Y \neq 1 \wedge Z \neq 1 \rightarrow Y' \neq 2 \wedge Z' \neq 2$. Now consider $(Y' : 1)$ and $(Z : 2)$. They are not allowed, and the network is no longer AC.

domains only the values that are AC, $D(X) = \{v\}$ and thus S is not returned by MAC+.
MAC($P{\times}P$) is complete: let S be a super solution, for any variables X, Y, let $v1$ be the value assigned to X in S, and $r1$ one of its possible repairs. Similarly $v2$ is the value assigned to Y and $r2$ its repair. It is easy to see that the pairs $\langle v1, r1\rangle$ and $\langle v2, r2\rangle$ are super arc-consistent, i.e, $\langle v1, v2\rangle$, $\langle v1, r2\rangle$ and $\langle r1, v2\rangle$ are allowed tuples. □

5 Extensions

Finding the Most Robust Solutions Often super solutions do not exist. First, from a theoretical perspective, the existence of a backbone variable guarantees that super solutions cannot exist. Second, from an experimental perspective (see next section), it is quite rare to have super solutions in which *all* variables can be repaired. To cure both problems, we propose finding the "most robust" solution that is as close as possible to a super solution.

For a given solution S, a variable is said to be *repairable* iff there exists at least a value v in its domain different from the one assigned in S, and v is compatible with all other values in S. The most robust solution is a solution where the number of repairable variables is maximal. Such a robust solution is guaranteed to exist if the problem is satisfiable. In the worst case, none of the variables are repairable. We hope, of course, to find some of the variables are repairable. For example, our experiments show that satisfiable instances at the phase transition and beyond have a core of roughly $n/5$ repairable variables.

To find the most robust solutions, we propose a branch and bound algorithm. The algorithm implemented is very similar to MAC+ (see 4.2), where AC is established on the non-assigned as well as on the assigned variables. The current lower bound computed by the algorithm is the number of singleton domains. The initial upper bound is n. Indeed, each singleton domain corresponds to an *un-repairable* variable, since no other value is consistent with the rest of the solution. The rest of the algorithm is a typical branch and bound procedure. The

	MAC+	MAC on $P{+}P$	MAC on $P{\times}P$	super MAC
$\langle 50, 15, 0.08, 0.5\rangle$				
CPU time (s)	788	43	53	**1.8**
backtracks	152601000	111836	**192**	2047
time out (3000 s)	12%	**0%**	**0%**	**0%**
$\langle 100, 6, 0.05, 0.27\rangle *$				
CPU time (s)	2257	430	3.5	**1.2**
backtracks	173134000	3786860	**619**	6487
time out (3000 s)	66%	7%	**0%**	**0%**

Fig. 6. Results at the phase transition. ($*$ only 50 instances of this class were given to MAC+)

first solution (or the proof of unsatisfiability) needs exactly the same time as the underlying MAC algorithm. Afterward, it will continue branching and discovering more robust solutions. It can therefore be considered as an *incremental anytime algorithm*. We refer to this algorithm as super Branch & Bound.

Optimization Problems For optimization problems, the optimal solution may not be a super solution. We can look for either the most repairable optimal solution or the super solution with the best value for the objective function. More generally, an optimization problem then becomes a *multi-criterion* optimization problem, where we are optimizing the number of repairable variables and the objective function.

6 Experimental Results

We use both random binary CSPs and job shop scheduling problems. Random CSP instances are generated using the 4 parameters $\langle n, m, p_1, p_2\rangle$ of Bessière's generator [1], where n is the number of variables, m is the domain size, p_1 is the constraint density, and p_2 is the constraint tightness. The job shop scheduling problem consist of n jobs and m machines. Each job is a sequence of activities where each activity has a duration and a machine. The problem is satisfiable iff it is possible to schedule the activities such that their order is respected and no machine is required by two activities that overlap, within a given makespan mk. Instances were generated with the generator of Watson *et al.* [17]. We define an instance with five parameters $\langle j, m, d_{min}, d_{max}, mk\rangle$ where j is the number of jobs, m the number of machine, d_{min} the minimum duration of an activity, d_{max} the maximum duration and mk the makespan. The actual duration of any activity is a random number between d_{min} and d_{max}.

Comparison: We compared the different solution methods using two samples of 100 random instances of the classes $\langle 50, 15, 0.08, 0.5\rangle$ and $\langle 100, 6, 0.05, 0.27\rangle$ at the phase transition. We observe, in Figure 6, that MAC on $P{\times}P$ prunes most, but is not practical when the domain size is large. As the problem size increases, super MAC outperforms all other algorithms in terms of runtimes.

Constrainedness and Hardness: We locate the phase transition of finding super solutions both experimentally and by an approximation based on the *κappa* framework [10].

Empirical Approach: We fixed $n = 40$ and $m = 10$ and we varied p_1 from 0.1 to 0.9 by steps of 0.02 and p_2 from 0.1 to 0.32 by steps of 0.012. For every combination of density/tightness, a sample of 100 instances were generated and solved by MAC and super MAC, with *dom/deg* as a variable ordering heuristic for MAC and (*super_dom*)/*deg* for super MAC. The number of visited nodes are plotted in Figure 7 (a) for MAC and in Figure 7 (b) for super MAC. As expected, the phase transition for super MAC happens earlier. Also, the phase transition peak is much higher (two orders of magnitude) for super CSP than for CSP.

Probabilistic Approximation Approach: For a CSP $P = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$ the expected number of solutions $\langle Sol \rangle$ is:

$$\langle Sol \rangle = (\prod_{x \in \mathcal{X}} m_x) \times (\prod_{c \in \mathcal{C}} (1 - p_c))$$

Where m_x is the domain size of x and p_c the tightness of the constraint c. A phase transition typically occurs around $\langle Sol \rangle = 1$ [10]. In our case, domain sizes and constraint tightness are uniform, therefore the formula can be simplified as follows:

$$1 = m^n \times (1 - p_2)^{(\frac{n(n-1)}{2} p_1)}$$

We assume that the $P \times P$ reformulation behaves like a random CSP. Note that $P \times P$ has one solution iff P has a single super solution. We can derive the values m' and $(1 - p_2')$ of the CSP $P \times P$:

$$m' =_{def} (m^2 - m)$$

Moreover, we can see $(1 - p_2)$ as the probability that a given tuple of values on a pair of constrained variables satisfies the constraint. For a given *pair* $\langle \langle v_1, v_2 \rangle, \langle w_1, w_2 \rangle \rangle$, this pair satisfies the reformulated constraint iff $\langle v_1, w_1 \rangle \in c$ *and* $\langle v_1, w_2 \rangle \in c$ *and* $\langle v_2, w_1 \rangle \in c$, which has a probability of $(1 - p_2)^3$. Hence, we have:

$$(1 - p_2') = (1 - p_2)^3$$

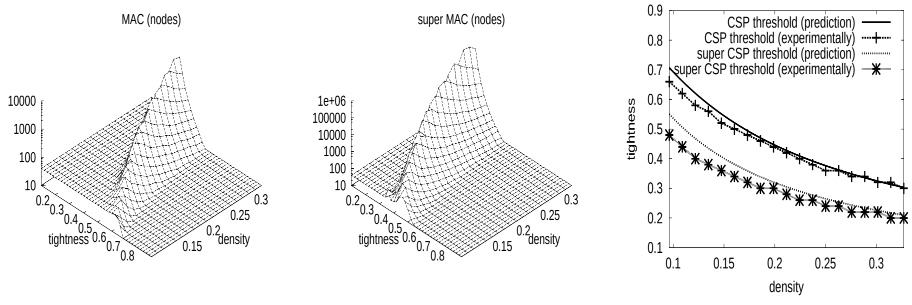
The formula thus becomes:

$$1 = (m^2 - m)^n \times (1 - p_2)^{3(\frac{n(n-1)}{2} p_1)}$$

In Figure 7 (c), we plotted those equations along with the values gathered from the empirical study. To do so, we considered, for every p_1, the minimal value p_2 such that the sample $\langle n, m, p_1, p_2 \rangle$ has more than half of its instances unsatisfiable. We observe that our approximations are very close to the empirical findings.

Job Shop Scheduling: We formulate the jobshop scheduling problem as a CSP, with one variable for each activity, and a domain size equal to the makespan mk minus its duration. We wish to minimize the makespan. We do so by iteratively increasing the makespan (mk) and solving the resulting decision problem. When a solution is found, we stop. We solved a sample of 50 problem instances for $j, m \in \{3, 4, 5\}$, $d_{min} = 2$ and $d_{max} \in \{10, 20, 30\}$[1]. Each sample was solved with

[1] In Figure 8, for every sample $m \times m$ the first three histograms stand for $d_{max} = 10$, the three following for $d_{max} = 20$, etc.



(a) Number of nodes visited by MAC

(b) Number of nodes visited by super MAC

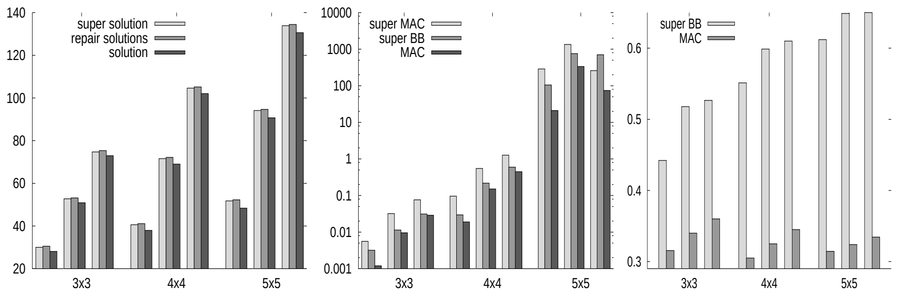
(c) Phase Transition Expectation

Fig. 7. Respective hardness to find a solution or a super solution

MAC, super MAC, and with super Branch & Bound. MAC and super Branch & Bound stopped at the same value of mk for which the problem is satisfiable. Whilst super MAC continued until the problem has a $(1, 0)$-super solution and then we optimize the makespan.

- Makespan: In Figure 8(a) we plot the makespan of the optimal solution found by MAC, the makespan of the optimal (1,0)-super solution returned by super MAC, and the worst possible makespan in case of a break to the optimal (1,0)-super solution. We observe that we have to sacrifice optimality to achieve robustness. Nevertheless, the increase in the makespan appears to be independent of the problem size and is almost constant.
- Search Effort: In Figure 8(b), we plot the time needed by MAC to find the optimal solution, by super MAC to find the optimal (1,0)-super solution, and by super Branch & Bound to find the most robust solution with the optimal makespan as found by MAC. As expected, more search is needed to find more robust solutions. Super MAC is on average orders of magnitude worse than MAC. Super Branch & Bound requires little effort for small instances, but much more effort when the problem size increases.
- Repairability: In Figure 8(c), we compare the percentage of repairable variable for the optimal solution found by MAC and the most robust optimal solution returned by super Branch & Bound. The optimal solutions returned by MAC have on average 33% of repairable variables, whilst the most repairable optimal solutions found by super Branch & Bound have 58% on average.

Minimal Core of Repairable Variables: We generated two sets of 50,000 random CSPs with 100 variables, 10 values per variable, 250 constraints forbidding respectively 56 and 57 tuples. Those problems are close to the phase transition, which is situated around 54 or 55 disallowed tuples, and at 58, no satisfiable



(a) Makespan (b) Search Effort (c) Repairability

Fig. 8. super solutions for the jobshop scheduling problem

instances were found among the 50,000 generated. The satisfiable instances (a total of 2407 for the first set, and 155 for the second) have in average 22% and 19% of repairable variables, respectively. The worst cases being 9% and 11%, respectively.

7 Related Work

Supermodels [12] and fault tolerant solutions [18] have been discussed earlier.

The notions of neighborhood interchangeability [7] and substitutability are closely related to our work, but whereas, for a given problem, interchangeability is a property of the values and works for all solutions, repairability is a property of the values in a given solution.

Uncertainty and robustness have been incorporated into constraint solving in many different ways. Some have considered robustness as a property of the algorithm, whilst others as a property of the solution (see, for example, dynamic CSPs [2] [11] [14], partial CSPs [6], dynamic and partial CSPs [13], stochastic CSPs [16], and branching CSPs [4]). In dynamic CSPs, for instance, we can reuse previous work in finding solutions, though there is nothing special or necessarily robust about the solutions returned. In branching and stochastic CSPs, on the other hand, we find solutions which are robust to the possible changes. However both these frameworks assume significant information about the likely changes.

8 Conclusion

To improve solution robustness in CP, we introduced the notion of super solutions. We explored reformulation and search methods to finding (1,0)-super solutions. We introduced the notion of super consistency, and develop a search algorithm, super MAC based upon it. Super MAC outperformed the other methods studied here. We also proposed super Branch & Bound, an optimization algorithm which finds the most robust solution that is as close as possible to a

(1,0)-super solution. Finally, we extended our approach to deal with optimization problems as well.

The problem of seeking super solutions becomes harder when multiples repairs are allowed, i.e, for $(1, b)$-super solutions. We aim to generalize the idea of super consistency to $(1, b)$-super solutions. In a similar direction, we would like to explore tractable classes of $(1, b)$-super CSPs. Furthermore, as with dynamic CSPs, we wish to consider the loss of n-ary no-goods and not just unary no-goods.

References

1. R. Dechter D. Frost, C Bessière and J.C. Régin. Random uniform CSP generator. url: http://www.ics.uci.edu/~dfrost/csp/generator.html, 1996.
2. A. Dechter and R. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings AAAI-88*, pages 37–42, 1988.
3. E. Hebrard B. Hnich and T. Walsh. Super CSPs. Technical Report APES-66-2003, APES Research Group, 2003.
4. D. W. Fowler and K. N. Brown. Branching constraint satisfaction problems for solutions robust under likely changes. In *Proceedings CP-00*, pages 500–504, 2000.
5. E. C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32:755–761, 1985.
6. E. C. Freuder. Partial Constraint Satisfaction. In *Proceedings IJCAI-89*, pages 278–283, 1989.
7. E. C. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proceedings AAAI-91*, pages 227–233, 1991.
8. J. Gaschnig. A constraint satisfaction method for inference making. In *Proceedings of the 12th Annual Allerton Conference on Circuit and System Theory.* University of Illinois, Urbana-Champaign, USA, 1974.
9. J. Gaschnig. Performance measurement and analysis of certain search algorithms. Technical report CMU-CS-79-124, Carnegie-Mellon University, 1979. PhD thesis.
10. I. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *Proceedings AAAI-96*, pages 246–252, 1996.
11. N. Jussien, R. Debruyne, and P. Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Proceedings CP-00*, pages 249–261, 2000.
12. A. Parkes M. Ginsberg and A. Roy. Supermodels and robustness. In *Proceedings AAAI-98*, pages 334–339, 1998.
13. I. Miguel. *Dynamic Flexible Constraint Satisfaction and Its Application to AI Planning.* PhD thesis, University of Edinburgh, 2001.
14. T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problem. *IJAIT*, 3(2):187–207, 1994.
15. T. Walsh. SAT v CSP. In *Proceedings CP-2000*, pages 441–456, 2000.
16. T. Walsh. Stochastic constraint programming. In *Proceedings ECAI-02*, 2002.
17. J.P. Watson, L. Barbulescu, A.E. Howe, and L.D. Whitley. Algorithms performance and problem structure for flow-shop scheduling. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 688–695, 1999.
18. R. Weigel and C. Bliek. On reformulation of constraint satisfaction problems. In *Proceedings ECAI-98*, pages 254–258, 1998.