Robust Solutions for Constraint Satisfaction and Optimization

Emmanuel Hebrard and **Brahim Hnich** and **Toby Walsh** [1,2]

Abstract. Super solutions are a mechanism to provide robustness to constraint programs [10]. They are solutions in which, if a small number of variables lose their values, we are guaranteed to be able to repair the solution with only a few changes. We extend the super solution framework along several dimensions to make it more useful practically. We present the first algorithm for finding super solutions in which the repair can, if needed, change variables that have not broken. We also extend the framework and algorithms to permit a wide range of practical restrictions on the breaks and repairs (for example, repairs might have to be later in time). We also show how to deal with symmetry when finding super solutions. Symmetry is a frequent problem in constraint solving. Experimental results suggest that it is even more important to tackle symmetry when looking for super solutions. Finally, we present results on job shop scheduling problems which demonstrate the tradeoff between solution robustness and makespan. For example, we are able to return solutions which are significantly more robust with no sacrifice in the makespan.

1 Introduction

Many decision and optimization problems contain uncertainty and thus, the user may require *robust* solutions. A solution is usually seen as robust if future changes are unlikely to affect it. It is difficult, however, to characterize such robustness whilst taking no assumption on the likelihood of the changes themselves. We consider here a slightly different definition of the notion of robustness, where the effect of certain changes on the solution can be bounded a priori. For example, when, in a scheduling problem, a machine breaks down or an activity is delayed, we would like to be able to repair it with a few local changes. Super solutions are a mechanism to guarantee such solution robustness within constraint programming [10]. 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. Considering only value removal might appear restrictive, however, no-good addition may be approximated by removing one value of the no-good. Alternatively, in the hidden representation, adding a no-good is equivalent to removing the corresponding value. On the other hand, we are not interested in 'positive' changes which can only add solutions. In [10], the authors explored in depth the simplest case, namely finding $(1, 0)$-super solutions.

In this paper, we extend this framework along a number of important dimensions to make it more useful and practical. First, we propose a novel search algorithm for finding (1,b)-super solutions. Second, we extend the super solution framework and this algorithm to deal with a wide range of practical restrictions on breaks and repairs. For instance, if values represent time, we might insist that all repairs use larger values. We might also identify only certain variables as brittle, or certain values as robust, etc. Third, as problems may not have any $(1, b)$-super solution, we propose a branch and bound algorithm to find the most robust solution closest to a $(1, b)$-super solution. Fourth, we study how to deal with symmetry while seeking super solutions. Finally, we present results on job shop scheduling problems which demonstrate the tradeoff between solution robustness and makespan.

2 Background

A *constraint satisfaction problem* (CSP) is a set of variables, each with a finite domain of values, and a set of constraints. A constraint is a relation defining the allowed values for these variables. A solution to a constraint satisfaction problem is an assignment of values to variables that satisfies all the constraints.

In [10], the authors introduced to constraint programming the notion of (a, b)-super solutions. Super solutions are a generalization of both fault tolerant solutions [17] and super models [9]. A solution is an (a, b)-super solution iff it is a solution and 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. The *alternative value* taken by a variable after it breaks is chosen from the other values in its domain. The *break set* is the set of variable that may break, and the *repair set* is the set of variables that can be used as repair. In general, deciding if a CSP has an (a, b)-super solution is NP-complete for any fixed a [10]. Not much is known about tractable classes. On binary CSPs with Boolean variables, finding (a, b)-super solutions is polynomial for $b \leq 1$ and NP-hard otherwise [13]. Unfortunately, as the following result shows, the relationship between the tractability of finding solutions and that of finding super solutions is not simple.

Theorem 1 *There exist classes of CSPs for which* SOLUBILITY *is polynomial but* (a, b)-SUPER SOLUBILITY *is NP-hard.*

Proof Suppose we modify a CSP by adding an extra value to each variable, and relaxing the constraints to permit variables to take this new value. The new CSP has all the solutions of the old, plus the solution that assigns the new value to each variable. As it always has at least one solution, SOLUBILITY is polynomial. However, (a, b)-SUPER SOLUBILITY is NP-hard as it requires finding the super solutions of the original CSP. □

[1] Cork Constraint Computation Centre, University College Cork, and Department of Information Science, Uppsala University (B. Hnich and T. Walsh). email: {e.hebrard,b.hnich,tw}@4c.ucc.ie

[2] All supported by Science Foundation Ireland and an ILOG grant.

Theorem 2 *There exist classes of CSPs for which* SOLUBILITY *is NP-complete but* (a, b)-SUPER SOLUBILITY *is polynomial.*

Proof Suppose we add to any CSP a variable with a singleton domain and no constraints on it. SOLUBILITY is still NP-complete. However, any break involving this variable is unrepairable. Such a problem has no (a, b)-super solution for any a or b. Thus (a, b)-SUPER SOLUBILITY is trivially polynomial. □

3 Finding $(1, b)$-super solutions

In [10], the authors only studied how to find $(1, 0)$-super solutions. Unfortunately, $(1, 0)$-super solutions will often not exist. Breaking a variable may necessarily require other variables to change. It is more practical to look for $(1, b)$-super solutions. We present therefore the first algorithm to find $(1, b)$-super solutions. Extending this algorithm to find (a, b)-super solutions is straightforward, but requires space that is exponential in a. The main idea is to extend an existing CSP algorithm with additional data structures for the repairs as well as procedures to ensure that any solution is also a $(1, b)$-super solution. In this case, `repair-MAC` extends the well known Maintaining Arc Consistency algorithm (MAC) [7, 8].

Algorithm 1: `repair-MAC`($P = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}, b$)

Data : $P = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}, b$
Result : S: a super solution and R: the set of repair solutions
$S \leftarrow \emptyset$ /* set of pairs $\langle var : va; \rangle$ */;
$Past \leftarrow \emptyset$ /* ordered set of variables */;
foreach $x \in \mathcal{X}$ **do**
  foreach $y \in \mathcal{X}$ **do**
    $R_x[y] \leftarrow min(\mathcal{D}(y))$;
`backtrack`($P, S, Past, R, b$, 0);

Procedure `backtrack` ($P, S, Past, R, b, n$) : Boolean

if $\mathcal{X} = Past$ **then** return True;
choose $x \in \mathcal{X} - Past$;
$Past[n] \leftarrow x$;
foreach $v \in \mathcal{D}(x)$ **do**
1   save $\mathcal{D}$ and R;
    $S \leftarrow S \cup \{\langle x : v \rangle\}$;
2   **if** AC(P) & $\forall y \in Past$, `repairable`($P, S, Past, R_x, 0, b$) **then**
      if `backtrack`($P, S, Past, R, b, lvl + 1$) **then** return True;
3   restore $\mathcal{D}$ and R;
    $S \leftarrow S - \{\langle x : v \rangle\}$;
$Past[n] \leftarrow nill$;
return False;

Procedure `repairable` ($P, S, Past, R_x, l, b$) :Boolean

if $l = |S|$ **then** return True;
$y \leftarrow Past[l]$;
1 **for** $v \leftarrow R_x[y]$ ***to*** $max_{init}(\mathcal{D}(y))$ **do**
2   **if** $x \neq y$ *or* $S[x] \neq v$ **then**
      $R_x[y] \leftarrow v$;
      if `check-repair`($P, S, Past, R_x, l, b$) **then**
        if `repairable`($P, S, Past, R_x, l + 1, b$) **then** return True;
3 $R_x[y] \leftarrow min(\mathcal{D}(y))$;
return False;

A repair solution R_x is associated with every variable x. $R_x[y]$ is the value assigned to y in the repair solution for x while $S[x]$ is the value assigned to x in the solution. The `backtrack` procedure tries to extend the current partial assignment, and backtracks for one of two reasons: we cannot extend the current partial assignment and satisfy the problem constraints (classical MAC), or the current partial assignment cannot be part of a $(1, b)$-super solution. When the current variable breaks, if all other partial assignments satisfying the constraints have more than b different values, then this partial assignment cannot be part of a super solution.

Procedure `check-repair` ($P, S, Past, R_x, l, b$) :Boolean

$dif = 0$;
1 **for** $i = 0$ ***to*** l **do**
    $y \leftarrow Past[i]$;
    if $x \neq y$ & $R_x[y] \neq S[y]$ **then** $dif \leftarrow dif + 1$;
if $dif > b$ **then** return False;
2 return consistency of the l first values in R_x;

The procedure `repairable` searches for partial repair solutions using backtracking, starting from the last repair found. We make sure that the alternative value used is different from the broken value, and the repair variable is different from the broken one (line 2). When backtracking, the value for the current level is reset to $min(D(x))$ in order to explore completely all the values at this level (line 3). In the worst case, `repairable` tries $O(n^{b+1}d^{b+1})$ repair solutions, which is polynomial for fixed b. One refinement to `repairable` (which does not require us to maintain domains for each repair) is to add some forward checking into the future. This can detect earlier domain wipe-outs. Finally, `check-repairs` checks two conditions: within each partial repair solution, no more than b variables (other than the broken variable) are assigned different values, and the partial repair solution is consistent with the problem constraints.

We show that this algorithm terminates, is sound and complete.

Theorem 3 `repair-MAC` *terminates and is sound and complete.*

Proof (Sketch)

Termination: immediate as the algorithm never revisits any partial assignment or repair for a particular break.

Soundness: the truth of the following is invariant. $\forall x \in Past$, *R_x is the first (in the lexicographical order) repair solution of S for x in the problem restricted to $Past$.*

Completeness: MAC is complete, therefore no partial assignment is omitted before checking for repairability. The check for repairability starts from the last repair found for that value. However, no assignment before this last repair in the search tree can be extended to the current variable since for all of them more than b changes were already done. □

4 Break and repair restrictions

In practice, we may have restrictions on how the problem is likely to break, or how we may repair it. We therefore extend the super solution framework to deal with such restrictions.

Break and repair set restrictions: A job shop problem may have some machines that are reliable and others that are not. If variables represent machines, then we can limit breaks to a subset of the variables. Similarly, whilst some activities may be reallocated, others may have to occur exactly when they are originally scheduled. If variables represent activities and values their times, then we need to limit repairs to a subset of the variables. It is straightforward to modify the algorithm to handle both such restrictions. For example, to deal with repair set restrictions, we add to `check-repair` a test that the b possible changes are within the repair set.

Alternative value restrictions: When a variable breaks, there are often restrictions on the alternative value that it can take. For example, when the values represent time, then an alternative value might have to be larger than the broken value. The algorithm can easily be modified to cope with such situations. In `repairable` (line 2), we change the test $S[x] \neq v$ to $S[x] \preceq v$, where $\preceq$ is any binary relation on the broken and alternative values.

Value and variable repair restrictions: We will often have restrictions on the repair allowed. We consider two common types of restrictions. First, the value repair restriction ensures that the variables repaired have a value, before they are repaired, which is larger than the smallest broken value, and all repair values are larger than the smallest broken value. Such a restriction is useful when, for instance, the values are times and we can only change events in the future. Second, the variable repair restriction ensures that all b repair variables are later in some ordering than the smallest of the a broken variables. Such a restriction is useful when, for instance, the variables are in a temporal order and we can only repair future variables. Again, it is easy to modify the algorithm to enforce either of these restrictions. For example, to add the value repair restrictions, we modify `check-repair` to test $\forall y \cdot R_x[y] \geq S[y]$.

Robust values: Often certain values may not be brittle and so cannot break. In addition, if certain values are chosen, they cannot be changed. It is again easy to modify the algorithm to deal with such robust values. For example, we modify `backtrack` so that it only finds repairs for those variables in the past whose values are brittle.

None of these restrictions changes the problem's complexity.

Theorem 4 (a, b)-SUPER SOLUBILITY *with break or repair set restriction, with alternative value restrictions, with value or variable repair restrictions, and with robust value restrictions is NP-complete for fixed* a*.*

Proof (Sketch) We show that the NP-completeness proof for the unrestricted problem in [4] can be extended to the restricted case. Since checking those restrictions is polynomial, we can use the same polynomial witness (the super solution plus the repair values).

To prove NP-hardness, binary CSP is reduced to super CSP. The domains of the variables are duplicated and the constraints are extended to behave equivalently on the duplicated (primed) values. This problem is satisfiable iff the orginal is. Moreover, if there exists a solution, then there exists a super solution since any set of a values can be primed (or unprimed) without affecting its validity.

This property continues to hold if we restrict the breaks to any subset of variables, we restrict the repairs to any subset of variables, we restrict the alternative values so that the only alternative to an unprimed value is the equivalent primed value, we order primed values after unprimed ones and restrict the repair values to later in this order, we put any ordering on the variables and restrict the repair variables to later in this order, or we make certain of the values robust against break or repair. □

Whilst a must be fixed, b need not be. Thus $(1, n/2)$-SUPER SOLUBILITY with any of the break or repair restrictions is NP-complete but $(n/2, 1)$-SUPER SOLUBILITY is not. If a is not fixed, (a, b)-SUPER SOLUBILITY with any of the restrictions is in PSPACE.

5 Optimization

Optimizing repairability: When $(1, b)$-super solutions do not exist, we might want to maximize the number of *repairable* variables. For a given solution S, a variable is said to be *repairable* iff there exists another solution S' which assigns the variable a different value, and S' differs in at most $b + 1$ variables from S. This gives a robust solution closest to a $(1, b)$-super solution. In a similar way to [10], we can adapt the `repair-MAC` algorithm to solve this optimization problem using a branch & bound scheme. The size of the break set less the number of breaks that cannot be extended is an upper bound on the number of repairable variables. We can adapt this algorithm to return the solution which can be repaired with minimum perturbation by keeping an upper bound on the maximum repair size. Other metrics are also possible like the average repair size.

Robustness and optimality: The optimal solution may not be a $(1, b)$-super solution. As in [10], we can either seek the most robust solution or the $(1, b)$-super solution with the best (but sub-optimal) objective function. More generally, an optimization problem turns into a *multi-criterion* optimization problem, where we optimize the number of repairable variables and the objective function.

6 Symmetry

Many real world problems contain symmetry [4]. For example, in a job shop scheduling problem, some of the machines may be identical. Similarly, some of the jobs may be identical. Such symmetry may increase the search space, as we can permute machines and jobs that are symmetric. Initial results suggest that dealing with symmetry is even more important when searching for super solutions than when searching for solutions. However, symmetry breaking methods need to be modified when looking for super solutions.

We identify three classes of symmetries: the class of *all* CSP symmetries, the class of symmetries generally seen in CSPs, and the class of symmetries that preserve super solutions. Given a CSP $P = \{\mathcal{X}, \mathcal{D}, \mathcal{C}\}$, we denote $\mathcal{U}$ the set containing all possible sets of pairs $\langle x, v\rangle$ where $x \in \mathcal{X}$ and $v \in \mathcal{D}(x)$, $\mathcal{I}$ the set containing only proper assignments in $\mathcal{U}$, $\mathcal{I}_c$ the set containing only consistent elements of $\mathcal{I}$, and S the set of solutions, i.e., the subset of $\mathcal{I}_c$ which elements are of size $|\mathcal{X}|$. An automorphism is a bijective mapping from a set to itself.

Definition A symmetry is an automorphism σ on $\mathcal{U}$ that preserves solutions, i.e., $I \in S \Rightarrow \sigma(I) \in S$. Σ is the set of all such symmetries.

In general, such symmetries do not preserve super solutions. We can identify two classes of symmetry that do.

Definition A decomposable symmetry is an automorphism γ on $\mathcal{U}$ which preserves consistency, i.e., if $I \in \mathcal{I}_c \Rightarrow \sigma(I) \in \mathcal{I}_c$, that also distributes over set union, i.e., $\forall x, y \in U, \gamma(x \circ y) = \gamma(x) \circ \gamma(y)$. Γ is the set of all such symmetries.

An example of a decomposable symmetry is the 90^o rotational symmetry of a row-wise representation of the n-queens problem. An even more restricted and useful class of symmetry that captures variable and value symmetries [11] is the following.

Definition A *strong* decomposable symmetry is an automorphism δ over $\mathcal{U}$, which preserves both consistency, i.e., if $x \in \mathcal{I}_c$ then $\delta(x) \in \mathcal{I}_c$ and validity, i.e., if $x \in \mathcal{I}$ then $\delta(x) \in \mathcal{I}$, that also distributes over set union. Δ is the set of all such symmetries.

It is easy to see that $\Delta \subset \Gamma \subset \Sigma$. We prove that if γ is a decomposable symmetry and I is a super solution then $\gamma(I)$ is also a super solution.

Theorem 5 *Given a CSP with a decomposable symmetry* γ*, then* γ *maps any* (a, b)*-super solution onto an* (a, b)*-super solution, and any assignment that is not an* (a, b)*-super solution onto another assignment that is not an* (a, b)*-super solution.*

Proof (Sketch) Suppose S is an (a, b)-supersolution, A is the set of all possible breaks of size at most a, R_i is the repair solution associated with each break $i \in A$. By definition, $\gamma(S)$ is also a solution.

We now show that $\gamma(A)$ is the set of all possible breaks of $\gamma(S)$. As γ is bijective, the size of A is equal to the size of $\gamma(A)$. We also have that, for each $i \in A$, $i \subseteq S$. Since, γ is decomposable, we also have that $\gamma(i) \subseteq \gamma(S)$ for each $\gamma(i) \in \gamma(A)$. Thus, $\gamma(A)$ is the set of all possible breaks (of size at most a) of $\gamma(S)$. The image of each repair solution, $\gamma(R_i)$, differs from $\gamma(S)$ in at most $a + b$ positions. Hence we have a repair solution for $\gamma(S)$ for each possible break in $\gamma(A)$. Hence, $\gamma(S)$ is an (a, b)-super solution. □

Symmetry breaking tools which eliminate symmetric solutions have to be modified when we look for super solutions. We highlight the subtleties through a simple example. Consider a CSP with two variables $X, Y \in \{1, 2\}$, and the following allowed tuples: $\{\langle 2, 2\rangle \langle 1, 2\rangle \langle 2, 1\rangle\}$. This problem has 3 solutions, but only one $(1, 0)$-super solution, $\langle 2, 2\rangle$. Now X and Y are symmetric. We can break this symmetry either during search or statically. We might add an ordering constraint between X and Y like $X \leq Y$. By doing so, we eliminate the solution $\langle 2, 1\rangle$ because it is symmetric to $\langle 1, 2\rangle$. However, we also lose a repair solution, $\langle 2, 1\rangle$, which is crucial for proving that $\langle 2, 2\rangle$ is a $(1, 0)$-super solution. Thus, by breaking the symmetry in a super CSP, we may loose some super solutions. One simple solution to the above problem is to ignore symmetry breaking techniques when checking for the consistency of the repair solutions.

7 Application

To demonstrate the practicality of this extended framework and to explore the tradeoff between robustness and optimality, we ran experiments on a larger number of job shop scheduling problems. Each problem consists of n jobs and m machines. Each job is a sequence of m activities, where each activity has a duration for its execution on one of the m machines. The objective is to schedule the activities such that their order is respected, no machine is required by two activities that overlap, and the makespan mk is minimized. Unfortunately, machines break down, requiring that the remaining activities be re-scheduled. We show that we can find robust solutions that are less sensitive to such changes.

We formulate the job shop scheduling problem as a CSP, with one variable for each activity, and a domain size equal to the makespan mk minus its duration. To find the minimal makespan, we start with mk equal to a lower bound and increase it till a solution exists. The lower bound is the larger of the length of the longest job, or the largest schedule for a single machine without slack. A schedule for a single machine without slack is simply the sum of all activities that must be executed on that machine plus the sum of the durations of activities that must execute before the first or after the last. We generate random job shop scheduling instances with the generator described in [14]. An instance has three parameters $\langle j, m, d_{max}\rangle$ where j is the number of jobs, m the number of machine, and d_{max} the maximum duration of an activity. The actual duration of any activity is a random number between 2 and d_{max}. We also consider the following extra restrictions:

- **break set:** this is either all activities associated with only *one* machine or with *all* the machines. In Tables 1 and 2, we use "1" and "all" to refer to these two cases.
- **alternative and repair value:** the alternative value v for the broken value w should satisfy $w + k < v$ where k is the expected duration for repairing a machine. In Tables 1 and 2, we put 2 or 4 opposite the entry *alternative* to denote the value of k. Similarly, we restrict repair values to be larger.
- **repair set:** this is either all activities or only the activities associated with one job. In Tables 1 and 2, we use "1" and "all" to these two cases.

break set	1				all			
alternative	2		4		2		4	
$\langle 3, 3, 10\rangle$	1.10		1.30		1.20		1.55	
$\langle 3, 3, 20\rangle$	1.05		1.16		1.11		1.31	
$\langle 3, 4, 10\rangle$	1.09		1.30		1.16		1.57	
$\langle 3, 4, 20\rangle$	1.04		1.16		1.9		1.31	
repair set	1	all	1	all	1	all	1	all
$\langle 3, 3, 10\rangle$	1.08	1.06	1.17	1.14	1.23	1.17	1.43	1.33
$\langle 3, 3, 20\rangle$	1.04	1.03	1.08	1.07	1.12	1.19	1.24	1.19
$\langle 3, 4, 10\rangle$	1.08	1.06	1.15	1.12	1.25	1.17	1.49	1.35
$\langle 3, 4, 20\rangle$	1.04	1.03	1.08	1.07	1.13	1.09	1.26	1.21

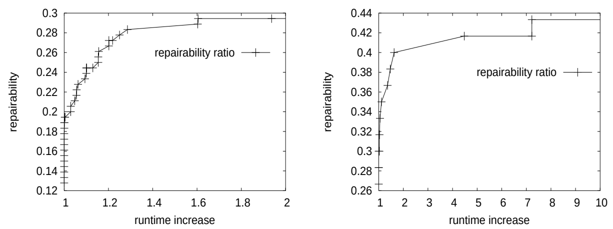
Table 1. Optimal makespan of super solutions/optimal makespan of solutions, upper table: $(1, 0)$, lower table: $(1, 2)$.

Optimal robust solutions: In Table 1, we report the percentage increase in the optimal makespan for $(1, b)$-super solutions with the different restrictions and for four different instance sizes. Each result represents the mean of 25 instances. We find $(1, 0)$-super solutions in the first four rows, and $(1, 2)$-super solutions in the next four rows. We observe that we have to sacrifice optimality to achieve robustness. Nevertheless, the increase in the makespan does not appear to be correlated with the problem size, but rather with the choice of parameters. In particular, the alternative value restrictions have the biggest influence.

Most robust optimal solutions: If the user wants to keep the optimal makespan, we still can optimize robustness using the $(1, b)$`-super Branch&Bound` algorithm. Table 2 shows we can increase the number of repairable variables in an optimal solution by between 50% to 250% without increasing the makespan. The main factor influencing this factor appears to be the size of the break set. In Figure 1 we show how much more search is required in order to increase the solution robustness of an optimal solution. The Y axis gives the fraction of repairable variable among all those subject to a break whilst X axis gives the increase in the runtime. We can see that a gain of about 50% in repairability can be obtained in a negligible increase of runtime.

break-set	1				all			
break size	2		4		2		4	
repair-set	1	all	1	all	1	all	1	all
$(1, 0)$	2.44		2.4		1.56		1.48	
$(1, 1)$	2.66	2	2.4	1.85	1.81	1.91	2.03	3.41
$(1, 2)$	2.66	2.06	2.4	3.6	1.83	2.56	2.48	-

Table 2. Ratio of repairable variables in the most robust optimal solution compared to number of repairable variables in the first optimal solution returned by MAC for 5 machines and 4 jobs problems.

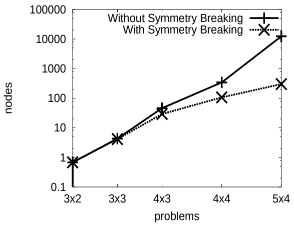


(a) $(1, 0)$-repairability, (3 machines are likely to break).

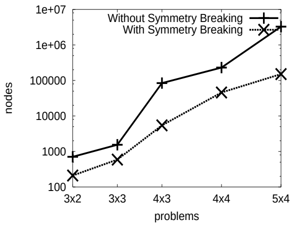
(b) $(1, 1)$-repairability, (1 machine is likely to break).

Figure 1. Search effort needed to improve robustness on a jobshop problem with 7 jobs and 6 machines.

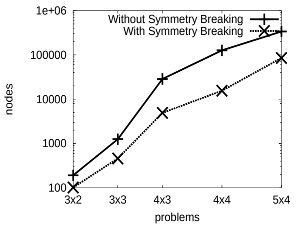
Breaking symmetry: We also ran experiments to determine the importance of breaking symmetry when looking for super solutions. We modified the generator so it could duplicate jobs. Duplicated jobs are equal in all respects. The corresponding activities (variables) can therefore be lexicographically ordered to break such symmetry. Note, however, that this ordering constraint is ignored in `check-repair`. We considered instances both with 3 symmetric jobs. Each of the 5 samples has 25 instances, with parameters $\langle 3, 2, 5\rangle$, $\langle 3, 3, 5\rangle$, $\langle 4, 3, 5\rangle$, $\langle 4, 4, 5\rangle$ and $\langle 5, 4, 5\rangle$ respectively. The break set is restricted to activities associated with two machines. In addition to the alternative value being greater than the broken value, two units of time are required to repair a machine. Finally, the repair set contains activities associated with two machines. Figure 2 presents the search effort saved by breaking symmetry. The figures show that breaking symmetry can reduce the search effort significantly. We note that the savings are more significant when looking for super solutions than when looking for solutions.



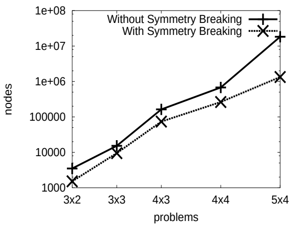
(a) Optimal Solution



(b) $(1, 0)$-super solution



(c) $(1, 1)$-super solution



(d) most robust optimal solution

Figure 2. The effect of symmetry breaking on search effort, for increasing problem size.

8 Related and future work

There have been a number of other mechanisms to deal with uncertainty and robustness in constraint solving. Robustness has been considered as a property of algorithms, as well as of solutions See, for example, dynamic CSPs [1], partial CSPs [6], dynamic and partial CSPs [12], mixed CSPs [2, 3], stochastic CSPs [16], and branching CSPs [5] as well as [15] for a comprehensive bibliography on this topic. In dynamic CSPs, for instance, we attempt to reuse previous search effort. However, the solutions returned are not robust in any sense. In stochastic CSPs, on the other hand, we find solutions which are robust to possible changes. However, unlike here, stochastic CSPs assume we have information about the probability of particular changes.

An important direction for future work is to identify ways to improve the efficiency of the `repair-MAC` algorithm. For example, can we exploit the fact that a repair solution is often a repair for multiple breaks? This is similar to the notion of multidirectionality when looking for supports in a AC algorithm. Moreover, if we look at the constraint graph, a repair variable must be within a short distance of a break. How do we best exploit this fact?

9 Conclusion

We have extended the super solution framework in several important dimensions to make it more useful and practical. 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. We have presented the first algorithms for finding $(1, b)$-super solutions, as well as the most robust solution closest to a $(1, b)$-super solution. We then extended the framework and algorithms to permit a wide range of practical restrictions on the breaks and repairs. In particular, we can place restrictions on the break set, the repair set, the alternative value used after a break, and the values and variables used in repairs. We also showed how to deal with symmetry when finding super solutions. Experimental results suggest that it is even more important to tackle symmetry when looking for super solutions than when looking for solutions. We also presented results on job shop scheduling problems which demonstrate the tradeoff between solution robustness and makespan. We saw that with a little extra effort, we are able to return solutions which are significantly more robust with no sacrifice in the makespan.

REFERENCES

[1] A. Dechter and R. Dechter, ‘Belief maintenance in dynamic constraint networks’, in *Proceedings AAAI’88*, pp. 37–42, (1988).
[2] H. Fargier and J. Lang, ‘Uncertainty in constraint satisfaction problems: a probalistic approach’, in *Proceedings ECSQARU’93*, pp. 97–104, (1993).
[3] H. Fargier, J. Lang, and T. Schiex, ‘Mixed constraint satisfaction: A framework for decision problems under incomplete knowledge’, in *Proceedings AAAI’96*, pp. 175–180, (1996).
[4] P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh, ‘Breaking row and column symmetries in matrix models’, in *Proceedings CP’02*, pp. 462–476, (2002).
[5] D. W. Fowler and K. N. Brown, ‘Branching constraint satisfaction problems for solutions robust under likely changes’, in *Proceedings CP’00*, pp. 500–504, (2000).
[6] E. C. Freuder, ‘Partial Constraint Satisfaction’, in *Proceedings IJCAI’89*, pp. 278–283, (1989).
[7] J. Gaschnig, ‘A constraint satisfaction method for inference making’, in *Proceedings of the 12th Annual Allerton Conference on Circuit and System Theory*.(1974).
[8] J. Gaschnig, ‘Performance measurement and analysis of certain search algorithms’, Technical report CMU-CS-79-124, Carnegie-Mellon University, (1979). PhD thesis.
[9] M. Ginsberg, A. Parkes, and A. Roy, ‘Supermodels and robustness’, in *Proceedings AAAI’98*, pp. 334–339, (1998).
[10] E. Hebrard, B. Hnich, and T. Walsh, ‘Super solutions in constraint programming’, in *Proceedings CPAIOR’04*, pp. 157–172, (2004).
[11] P. Meseguer and C. Torras, ‘Solving strategies for highly symmetric CSPs’, in *Proceedings IJCAI’99*, pp. 400–405, (1999).
[12] I. Miguel, *Dynamic Flexible Constraint Satisfaction and Its Application to AI Planning*, Ph.D. dissertation, University of Edinburgh, 2001.
[13] A. Roy and C. Wilson, ‘Supermodels and closed sets’, *Electronic Colloquium on Computational Complexity (ECCC)*, (2000).
[14] E. D. Taillard, ‘Benchmarks for basic scheduling problems’, *European Journal of Operational Research*, (1993).
[15] G. Verfaillie and N. Jussien, ‘Dynamic constraint solving’. Tutorial - online notes available http://www.emn.fr/jussien/CP03tutorial.
[16] T. Walsh, ‘Stochastic constraint programming’, in *Proceedings ECAI’02*, (2002).
[17] R. Weigel and C. Bliek, ‘On reformulation of constraint satisfaction problems’, in *Proceedings ECAI’98*, pp. 254–258, (1998).