Improved algorithm for finding (a,b)-super solutions

Emmanuel Hebrard
National ICT Australia and
University of New South Wales
Sydney, Australia.
ehebrard@cse.unsw.edu.au

Brahim Hnich
Cork Constraint Computation Centre
University College Cork, Ireland.
brahim@4c.ucc.ie

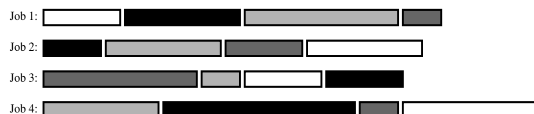
Toby Walsh
National ICT Australia and
University of New South Wales
Sydney, Australia.
tw@cse.unsw.edu.au

Abstract

In (EHW04b), the authors introduced to constraint programming the notion of (a, b)-super solutions. 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. This concept is useful for scheduling in dynamic and uncertain environments when the robustness of the schedule is a valuable property. We introduce a new algorithm for finding super solutions that improves upon the method introduced in (EHW04a) in several dimensions. This algorithm is more space efficient as it only requires to double the size of the original constraint satisfaction problem. The new algorithm also permits us to use any constraint toolkit to solve the master problem as well as the subproblems generated during search. We also take advantage of multi-directionality and of inference based on the *neighborhood* notion to make the search for a solution faster. Moreover, this algorithm allows the user to easily specify extra constraints on the repairs.

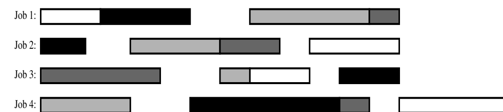
Introduction

In (EHW04b), the authors introduced to constraint programming the notion of (a, b)-super solutions. Super solutions are a generalization of both fault tolerant solutions (WB98) and super models (MGR98). These are solutions such that a small (bounded) perturbation on the input will have proportionally small repercussions on the outcome. For instance when solving a scheduling problem, we may want that, in the event of a machine breaking, or of a task executing longer than expected, the rest of the schedule should change as little as possible if at all. As a concrete example, consider the following job-shop scheduling problem, where we need to schedule four jobs consisting of four activities, each requiring a different machine. The usage of a machine is exclusive, and the sequence of a job is to be respected.

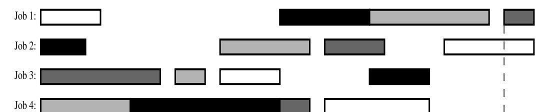


The second figure shows an optimal solution.

Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.



One may argue that this solution is not robust. Indeed activities are tightly grouped and a "break" on a machine, and the subsequent delay, may trigger further delays in the schedule. On the other hand the next figure shows a solution where, for a small makespan increase, a large proportion of activities can be delayed of two units of time, without affecting at all the rest of the schedule.



Such solutions are thus *stable* in the sense that when invalidated, a close alternative solution can be applied. However, there are a number of other ways to capture robustness. for instance in (HF93), the approach to robustness is probabilistic and a robust solution is simply one that is likely to remain valid after a contingent change. The problem of scheduling under uncertainty has been widely studied in the past (See (AD01) and (NPO04) for instance). We wish to investigate how super solutions compare to these specialized methods. For instance, if we consider the *slack-based* framework (AD01), the intuitive idea is that a local perturbation will be "absorbed" if enough slack is available, end therefore solutions involving slack are preferred. Now, one can think of a scenario where the best reaction to a delay or a break would not be to delay the corresponding (plus perhaps few other) activity, but to postpone it and advance the starting time of another activity instead. This situation is not captured by slack based method. Although, it is important to notice that if the latter approach aims at minimizing the impact of a delay on the makespan, it is only a secondary consequence for super solutions. Indeed, the main concern is to limit the number of activities to reschedule. Therefore, a direct comparison is difficult. The concept of *flexible schedule* is more closely related to our view of robustness as this method promotes stability against localized perturbations.

On the other hand, super solutions have a priori several drawbacks compared to such specialized approaches.

Firstly, finding super solutions is a difficult problem and as a result the methods proposed so far are often restricted to toy problems like the one used in the previous example.

Algorithms with better performance have been proposed for very restricted classes of super solution. However, if we do not restrict ourselves to these classes, solving a problem of reasonable size is often out of reach with the current approaches.

Another difficulty is that, being a general framework, it is not always immediately applicable to a specific problem. In (EHW04a) we showed how super solutions have to be adapted to cope with specifics of job shop scheduling problems in particular. For instance, if variables are activities and values are time points, then we cannot schedule to an earlier time point as a response to a break. Moreover, moving the same activity to the next consecutive time point may not be a valid alternative.

In this paper we introduce a new algorithm that can help to address the above drawbacks. The central feature of the new algorithm is that it is closer to a "regular" solver, we solve both the original problem as well as sub-problems generated during search using any standard constraint solver. Therefore, methods that have been proven to be effective in a particular area (like shaving (CP94), specialized variable orderings (SF96) or specialized constraint propagators) can be used both for the main problem and the sub-problems. We also propose a *more effcient* and *more effective* algorithm than what has been proposed in (EHW04a). The new algorithm is more efficient as we avoid solving *all* sub-problems and it is *more effective* by using the information gathered when solving these sub-problems to get more pruning on future variables. Moreover, this architecture also helps to adapt the criteria of robustness to the problem. Indeed, to model a particular requirement we can just add it as a constraint to the sub-problems.

Formal background and notations

A constraint satisfaction problem (CSP) P consists of a set of variables $\mathcal{X}$, a set of domains $\mathcal{D}$ such that $\mathcal{D}(X_i)$ is the finite set of values that can be taken by the variable X_i, and a set of constraints $\mathcal{C}$ that specify allowed combinations of values for subsets of variables. We use upper case for variables (X_i) and lower case for values (v). A full or partial instantiation $S = \{\langle X_1 : v_1 \rangle, \dots \langle X_n : v_n \rangle\}$ is a set of assignments $\langle X_i : v_j \rangle$ such that $v_j \in X_i$. We will use $S[i]$ to denote the value assigned to X_i in S. A (partial) solution is an instantiation satisfying the constraints. Given a constraint C_V on a set of variables V, a *support* for $X_i = v_j$ on C is a partial solution involving the variables in V and containing $X_i = v_j$. A variable X_i is *generalized arc consistent* (*GAC*) on C iff every value in $\mathcal{D}(X_i)$ has support on C. A constraint C is GAC iff each constrained variable is GAC on C, and a problem is GAC iff all constraints in $\mathcal{C}$ are GAC. Given a CSP P and a subset $A = \{X_{i_1}, \dots X_{i_k}\}$ of $\mathcal{X}$, a solution S of the restriction of P to A (denoted $P|_A$) is a partial solution on A such that if we restrict $\mathcal{D}(X_i)$ to $\{S[i]\}$ for $i \in [i_1..i_k]$, then P can be made GAC without domain wipe-out.

We introduce some notations used later in the paper. the function $H(S, R)$ is defined to be the Hamming distance between two solutions R and S, i.e., the number of variables assigned to different values in S and R. We also define $H_A(S, R)$ to be the Hamming distance restricted to the variables in A.

$$H_A(S, R) = \sum_{X_i \in A} (S[i] \neq R[i])$$

An *a-break* on a (partial) solution S is a combination of a variables among the variables in S. A *b-repair* of S is a (partial) solution R such that $H_A(S, R) = |A|$ and $H(S, R) \leq (a + b)$. In other words, R is an alternative solution for S such that if the assignments of the variables in A are forbidden, the remaining "perturbation" is restricted to b variables.

Definition 1 *A solution S is an (a, b)-super solution iff for every $a' \leq a$, and for every a'-break of S, there exists a b-repair of S.*

The basic algorithm

We first describe a very simple and basic version of the algorithm without any unnecessary features. Then we progressively introduce modifications to make the algorithm more efficient and more effective.

The basic idea is to ensure that the current partial solution is also a partial super solution. In order to do so, as many sub-problems as possible breaks for this partial solution have to be solved. The solutions to these sub-problems are partial repair solutions. We therefore work on a copy of the original problem that we change and solve for each test of *reparability*. Note that the sub-problem is much easier to solve than the main problem, for several reasons. The first reason is that each of the sub-problems is polynomial. Indeed, since a repair solution must have less than $a + b$ discrepancies with the main solution, the number of possibilities is bounded by $n^{a+b} d^{a+b}$. Typically, the cost of solving one such sub-problem will be far below this bound since constraint propagation is used. Another reason is that we can reuse the variable ordering dynamically computed whilst solving the main problem. Furthermore, we will demonstrate later that not all breaks have to be checked, and that we can infer inconsistent values from the process of looking for a repair. Pruning the main problem is critical, as it not only reduces the search tree, but also reduces the number of sub-problems to be solved.

The algorithm for finding super solutions is, in many respects, comparable to a global constraint. However, it is important to notice that we *cannot* define a global constraint ensuring that the solution returned is a super solution as this condition depends on the variables domains, whilst constraints should only depend on the values assigned to variables and not their domains. For example, consider two variables X_1, X_2 taking values in $\{1, 2, 3\}$ such that $X_1 < X_2$. The assignment $\langle X_1 = 1, X_2 = 3 \rangle$ is a $(1, 0)$-super solution, however, if the original domain of X_1 is $\{1, 3\}$, then $\langle X_1 = 1, X_2 = 3 \rangle$, is *not* a $(1, 0)$-super solution. Nevertheless, the algorithm we introduce could be seen as a global constraint implementation as it is essentially an oracle tightening the original problem It is however important to ensure that this oracle is not called for every change in a domain as it can be costly.

Initialization: In Algorithm 1, we propose an pseudo code for finding super solutions. The input is a CSP, i.e., a triplet $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ and the output a (a, b)-super solution S. We first create a copy P' of P, where $X'_i \in \mathcal{X}'$ iff $X_i \in \mathcal{X}$ and $C' \in \mathcal{C}'$ iff $C \in \mathcal{C}$. This copy will be used to find b-repairs. At any point in the algorithm, $\mathcal{D}(X_i)$ (resp. $\mathcal{D}'(X'_i)$) is the *current* domain of X_i (resp. X'_i). The set $Past \subseteq \mathcal{X}$ contains all variables that are already bound to a value and we denote $Past'$ the set containing the same variables, but "primed", $Past' = \{X'_i | X_i \in Past\}$.

Algorithm 1: `super-MAC`(P, a, b)

```
Data    : P, a, b
Result  : S: an (a, b)-super solution
S ← ∅; Past ← ∅; P' ← P;
if ¬backtrack(P, P', S, Past, a, b) then
    print "NO SUPER-SOLUTION";
print "A SUPER-SOLUTION FOUND: S";
```

Main Backtracker Procedure (Algorithm 2) searches and backtracks on the main problem P. It is in very similar to a classical backtracker that maintain GAC at each node of the search tree, except that we also add a call to the procedure `reparability` at each node. Note that any solver or local/global consistency property can be used instead as long as the procedure `reparability` is called. A possible way of implementing `reparability` –in a standard constraint toolkit– can be as a global constraint containing internally the extra data structure P' and an associated specialised solver. As such global constraint can be costly, it should not be called more than once per node.

Algorithm 2: `backtrack`$(P, P', S, Past, a, b)$: Bool

```
if Past = X then return True;
choose X_i ∈ X \ Past;
Past ← Past ∪ {X_i};
foreach v ∈ D(X_i) do
    save D;
    D(X_i) ← {v};
    S ← S ∪ {⟨X_i : v⟩};
    if AC-propagate(P) & reparability(P, P', S, Past, a, b)
    then
        if backtrack(P, S, Past, a, b) then return True;
    restore D;
    S ← S − {⟨X_i : v⟩};
Past = Past − {X_i};
return False;
```

Enforcing reparability: Procedure `reparability` (Algorithm 3) makes sure that each a-break of the solution S has a b-repair. If $|S| = k$ then we check all combinations of less than a variables in S, that is $\sum_{j \leq a} \binom{k}{j}$ breaks. This number has no closed form, though it is bounded above by k^a. For each a-break, we model the problem of the existence of a b-repair using P'. Given the main solution S and a break A, we need to find a b-repair, that is, a solution R of $P'|_{Past'}$ such that $H_A(S, R) = |A|$ and $H(S, R) \leq |A| + b$. The domains of all variables are set to their original state. Then for any $X'_i \in A$, we remove the value $S[i]$ from $\mathcal{D}'(X'_i)$, thus making sure that $H_A(S, R) = |A|$. We also add an ATMOSTkDIFF constraint that ensures $H(S, R) \leq k$, where $k = |A| + b$. Finally, we solve $P'|_{Past'}$, it is easy to see that any solution is a b-repair.

Algorithm 3: `repairability`$(P, S, Past, a, b)$:Bool

```
foreach A ⊆ Past' such that |A| ≤ a do
    foreach X_i ∈ A do
        D'(X'_i) ← D(X_i) − {S[i]};
    k ← (|A| + b);
    S' ← solve(P'|_Past' + ATMOSTkDIFF(X', S));
    if S' = nil then return False;
return True;
```

Propagating the ATMOSTkDIFF constraint: The ATMOSTkDIFF constraint is defined as follows:

Definition 2 ATMOSTkDIFF$(X'_1, \ldots X'_n, S)$ *holds iff* $k \geq \sum_{i \in [1..n]} (X'_i \neq S[i])$

This constraint ensures that the solution we find is a valid partial b-repair by constraining the number of discrepancies to the main solution to be lower than $a + b$. To enforce GAC on such a constraint, we first compute the smallest expected number of discrepancies to S. Since S is a partial solution we consider the *possible* extensions of S. Therefore, when applied to the auxiliary CSP P' this number is simply

$$d = |\{i | \mathcal{D}'(X'_i) \cap \mathcal{D}(X_i) = \emptyset\}|$$

We have three cases:

1. If $d < k$ then the constraint is GAC as every variable can be assigned any value providing that all other variables X'_i take a value included in $\mathcal{D}(X_i)$, and we will still have $d \leq k$.
2. If $d > k$ then the constraint cannot be satisfied.
3. If $d = k$ then we can set the domain of any variable X'_i such that $\mathcal{D}(X'_i) \cap \mathcal{D}(X_i) \neq \emptyset$ to $S[i]$.

Comparison with previous algorithm This new algorithm is simpler than the one given in (EHW04a) as no extra data structure is required for keeping the current state of a repair. Moreover, the space required is at most twice the space required for solving the original problem, whilst the previous algorithm stored the state of each search for a b-repair. We want to avoid such data structures as they are exponential in a. Even though a is typically a small constant, this can be prohibitive. Another advantage in doing so is that the search for a repair can easily be done, whereas in the previous algorithm, doing so would have been difficult without keeping as many solver states as breaks, since the search was starting from the point it ended in the previous call.

The search tree explored by this algorithm is strictly smaller than that explored by the previous algorithm. The

methods are very comparable as they both solve sub-problems to find a b-repair for each break. However, since the sub-problems solved by this previous algorithm were implemented as a simple backtrack procedure (without constraint propagation), it was not possible to check if a b-repair would induce an arc inconsistency in an unassigned variable.

Improvements

Now we explore several ways of improving the basic algorithm.

Repair multi-directionality

The multi-directionality is a concept used for instance for implementing general purpose algorithms for enforcing GAC. The idea is that a tuple is a support for every value it involves. The same metaphor applies when seeking repairs instead of supports. In our case, suppose that we look for a $(2,2)$-super solution and suppose that R is a repair solution for a break on the variables $\{X,Y\}$ that require reassigning the variables $\{V,W\}$. This constitutes also a repair for $\{X,V\},\{X,W\},\{Y,V\},\{Y,W\}$ and $\{V,W\}$. We therefore need not to look for repair for these breaks.

We used a simple algorithm from Knuth (Knu) to generate all $\leq a$-breaks. This algorithm generates the combinations in a deterministic manner, and therefore constitutes an ordering on these combinations. Moreover, this ordering has the nice property that given one combination in input, one can compute the rank of this combination in the ordering in linear time on the size of the tuple. The size of the tuple is in our case a small constant, we thus have an efficient way of knowing if the break that we currently consider is covered by an earlier repair. Each time a new repair is computed, all breaks it covers are added to a set, then when we generate a combination, we simply check that its index is not in this set otherwise we do not need to find a repair for it.

Neighborhood-based inference:

The second observation that we make to improve the efficiency is less obvious but has broader consequences. First, let us introduce some necessary notation:

A path linking two variables X and Y is a sequence of constraints $C_{V_1},\dots C_{V_k}$ such that $i=j+1 \Rightarrow V_i \cap V_j \neq \emptyset$ and $X \in V_1$ and $Y \in V_k$, k is the length of the path. The distance between two variables $\delta(X,Y)$ is the length of the shortest path between these variables ($\delta(X,X)=0$). $\Delta_d(X)$ denotes the neighborhood at a distance exactly d of X, i.e., $\Delta_d(X)=\{Y \mid \delta(X,Y)=d\}$. $\Gamma_d(X)$ denotes the neighborhood up to a distance d of X i.e., $\Gamma_d(X)=\{Y \mid \delta(X,Y)\leq d\}$. Similarly, we define the neighborhood $\Gamma_d(A)$ (resp. $\Delta_d(A)$) of a subset of variable A as simply $\bigcup_{X\in A}\Gamma_d(X)$ (resp. $\Delta_d(A)$).

Now we can state the following lemma which will be central to all the subsequents improvements. It shows that if there exists a b-repair for a particular a-break A, then all reassignments are within the neighborhood of A up to a distance b.

Lemma 1 *Given a solution S and a set A of a variables, the following equivalence holds:*

$$\exists R \textit{ s.t. } (H(S|_A,R|_A)=a \textit{ and } H(S,R)<d)$$
$$\Leftrightarrow$$
$$\exists R' \textit{ s.t. } (H(S|_A,R'|_A)=a \textit{ and }$$
$$H(S|_{\Gamma_{d-a}(A)},R'|_{\Gamma_{d-a}(A)})=H(S,R')<d)$$

Proof: We prove this lemma constructively. We start from two solutions S and R that satisfy the premise of this implication and construct R' such that S,R' satisfy the conclusion. We have $H(S,R)=k_1<k$, therefore exactly k_1 variables are assigned differently between S and R. We also know that $H(S|_A,R|_A)=|A|=a$ therefore only $b=k_1-a$ are assigned differently outside A. Now we change R into R' in the following way. Let d be the smallest integer such that $\forall X_i \in \Delta_d(A),\ R[i]=S[i]$. It is easy to see that $d\leq b$ as $\Delta_{d_1}(A)$ and $\Delta_{d_2}(A)$ are disjoint iff $d_1\neq d_2$. We let all variables in $\Gamma_d(A)$ unchanged, and for all other variables we set $R'[i]$ to $S[i]$. Now we show that R' satisfies all constraints. Without loss of generality, consider any constraint C_V on a set of variables V. By definition, the variables in V belongs to at most two sets $\Delta_{d_1}(A)$ and $\Delta_{d_2}(A)$ such that d_1 and d_2 are consecutive (or possibly $d_1=d_2$). We have 3 cases:

1. $d_1\leq d$ and $d_2\leq d$: all variables in V are assigned as in R, therefore C_V is satisfied.
2. $d_1>d$ and $d_2>d$: all variables in V are assigned as in S, therefore C_V is satisfied.
3. $d_1=d$ and $d_2=d+1$: the variables in $\Delta_{d_2}(A)$ are assigned as in S, and by definition of R', the variables in $\Delta_{d_1=d}(A)$ are assigned as in S, therefore C_V is satisfied.

$\diamond$

Computing this neighborhood is not time expensive, as it can be done as a preprocessing step. A simple breadth first search on the constraint graph, i.e., the graph were any two variables are connected iff they are constrained by the same constraint. The neighborhood $\Gamma_d(A)$ of a break A is recomputed each time, however it just requires a simple union operation over the neighborhood of the elements in A.

Updates of the auxiliary CSP: The first use of Lemma 1 is straightforward. We know that, for a given break A, there exists a b-repair only if there exists one that sharea all assignments outside $\Gamma_b(A)$. Therefore, we can make P' equal to the current state of P apart from $\Gamma_b(A)$. This does not make the algorithm stronger. However, we can then post an ATMOSTkDIFF constraint only on $\Gamma_b(A)$ instead of $\mathcal{X}'$, since we have all the pruning on $\mathcal{X}'\setminus\Gamma_b(A)$ for "free".

Avoiding useless repair checks: Now suppose that $\Gamma_{b+1}(A)\subseteq Past'$. Then we know that this break has already been checked at the previous level in the search tree, and the repair that we found has the property that assignments on $\Delta_{b+1}(A)$ are the same as in the current solution. Thus any extension of the current partial solution, is also a valid extension of this repair. Therefore we know that

this repair will hold in any subtree, hence we do not need to check it unless backtracking beyond this point.

Tightening the ATMOSTkDIFF **constraint:** Considering the property of Lemma 1, we can tighten the ATMOSTkDIFF constraint by forbidding some extra tuples. Doing this, we get a tighter pruning when doing arc consistency, while keeping at least one solution, if such solution exists. The first tightening is that all differences outside $\Gamma_b(A)$ are forbidden. But, we can do even more inference. For instance, suppose that for that we look for a 3-repair for the break $\{X'_1\}$ and that $\Delta_1(X'_1) = \{X'_2, X'_3\}, \Delta_2(X'_1) = \{X'_4\}, \Delta_3(X'_1) = \{X'_5,\}$, and the domains are as follows:

$$X'_1 = \{1,2\}, X'_2, X'_3 = \{3,4\}, X'_4, X'_5 = \{1,2,3,4\}$$

Moreover, suppose that for the main backtracker, the domains are as follows:

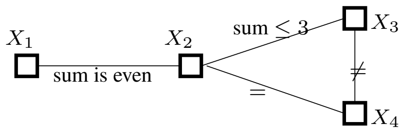
$$X_1 = 3, X_2, X_3, X_4, X_5 = \{1,2\}$$

We can observe that already 2 reassignments are made at distance 1 from X_1. As a consequence, if X'_5 was to be assigned differently to X_5, then X_4 would have to be equal to X_4, and therefore there is no discrepancy on any variable from $\Delta_2(X'_1)$, hence there must be a repair such that any variable outside $\Gamma_2(X_1)$ is assigned as in the main solution. We can thus prune the values 3 and 4 from $\mathcal{D}(X'_5)$ (to make it equal to X_5).

Inference from repair-seeking to the main CSP: We can infer that some values of the main CSP will never participate in a super solution while seeking for repairs. This allows us to prune the future variables, which can greatly speed up the search process, especially when combined with GAC propagation on "regular" constraints. For instance consider constraint problem P, composed of the domain variables:

$$X_1 = \{1,2,4\}, X_2 = \{1,2\}, X_3 = \{1,2\}, X_4 = \{1,2\}$$

subject to the following constraint network:



We have $P' = P$, and it is easy to see that P is arc consistent. Now suppose that we look for a $(1,1)$-super solution, and our first decision is to assign the value 1 to X_1. The domains are reduced so that P remains arc consistent:

$$X_1 = \{1\}, X_2 = \{1\}, X_3 = \{1,2\}, X_4 = \{1,2\}$$

Then we want to make sure that there exist a 1-repair for the break $\{X_1\}$. We then consider P' where $\mathcal{D}(X'_1)$ is set to $\mathcal{D}(X'_1) \setminus \{1\}$. Moreover the constraint ATMOST2DIFF is posted on $\Gamma_1(\{X_1\}) = (X_1, X_2)$:

$$X'_1 = \{2,4\}, X'_2 = \{1,2\}, X'_3 = \{1,2\}, X'_4 = \{1,2\}$$

Since $P'|_{\{X'_1\}}$ is satisfiable, (for instance, $\{\langle X'_1 : 2\rangle\}$ is a partial solution that does not produce a domain wipe out in any variable of P') we continue searching. However, if before solving P' in order to find a repair we first propagate arc consistency, then we obtain the following domains:

$$X'_1 = \{2,4\}, X'_2 = \{2\}, X'_3 = \{1\}, X'_4 = \{2\}$$

Observe that $2 \in \mathcal{D}(X_3)$ whilst $2 \notin \mathcal{D}(X'_3)$, this means that no repair for X_1 can assign the value 2 to X_3. However, Lemma 1 works in both direction, since $X'_3 \notin \Gamma_1(X'_1)$, we can conclude that X'_3 and X_3 should be equal, and therefore we can prune the value 2 directly from X_3. In this toy example, this removal will make P arc inconsistent, and therefore we can conclude without searching that X_1 cannot be assigned to 1.

Notice that this extra pruning comes at no extra cost, the only condition that we impose is to make P' arc consistent *before* searching on it. After this arc consistent preprocessing for a break A, any value pruned from the domain of a variable $X'_i \in \mathcal{X}' \setminus \Gamma_b(A)$, can be pruned from X_i as well.

The main drawback of this method is that as soon as the problem involves a global constraint (for instance "all the variables must be different"), then typically we have $\Gamma_1(X_i) = \mathcal{X}$ for any $X_i \in \mathcal{X}$. Therefore all previous improvements based on neighborhood are useless. However, one can make such inference, but using a different reasoning, even in the presence of large arity constraints. The idea is the following: Suppose that after enforcing GAC on P', the least number of discrepancies is exactly $a + b$, that is, $Diff = \{i | \mathcal{D}(X_i) \neq \mathcal{D}'(X'_i)\}$ & $|Diff| = a + b$. We can deduce that any variable X'_j such that $j \notin Diff$ must be equal to X_j, for *any* b-repair. Indeed, it applies to any repair, since only pre-processing and no search was used. Therefore, we can prune domains in both P and P' as follows:

$$\forall i \notin Diff\ \mathcal{D}(X_i) \leftarrow \mathcal{D}(X_i) \cap \mathcal{D}(X'_i)\ \&\ \mathcal{D}(X'_i) \leftarrow \mathcal{D}(X_i)$$

We can therefore modify `reparability` by taking into account the previous observations (Algorithm 4).

Extensions

In (EHW04a), the authors propose to extend or restrict super solutions in several directions to make them more useful practically. In scheduling problems, we may have restrictions on how the machines are likely to break, or how we may repair them. Furthermore, we have an implicit temporal constraint that forbid some reassignments. For instance, when a variable breaks, there are often restrictions on the alternative value that it can take. When the values represent time, then an alternative value might have to be larger than the broken value. Alternatively, or in addition, we may want the repair to be chosen among larger values, for some ordering. It may also be the case that certain values may not be brittle and so cannot break. Or that if certain values are chosen, they cannot be changed. This algorithm allows to express these restrictions (and many more) very easily, as they can be modelled as extra constraints on P'.

For instance to model the fact that an alternative value has to be larger than the broken value, one can post the unary

Algorithm 4: `repairability`$(P, S, Past, a, b)$:Bool

```
covered ← ∅;
foreach A ⊆ Past' such that |A| = a and A ∉ covered and Γ_{b+1} ⊈ Past' do
    foreach X_i ∈ A do
        D'(X'_i) ← D'(X'_i) − {S[i]};
    foreach X_i ∈ (X \ Γ_b(A)) do
        D'(X'_i) ← D(X_i);
    if ¬AC-propagate(P') then return False;
    Diff ← {i | D(X_i) ≠ D'(X'_i)};
    if Diff = a + b then
        foreach X_i ∈ (X \ Diff) do
            D(X_i) ← D'(X'_i) ← (D'(X'_i) ∩ D(X_i));
    foreach X_i ∈ (X \ Γ_b(A)) do
        D(X_i) ← D'(X'_i);
    S' ← solve(P'|_{Past'} + AtMost(|A| + b)Diff(Past', S));
    if S' = nil then return False;
    Diff ← {i | S'[i] ≠ S[i]};
    foreach A' ⊆ Diff such that |A'| = a do
        covered ← covered ∪ {A};
```

constraint $X > S[X]$, where X is any variable involved in the break, and $S[X]$ is the value assigned to this variable in the main solution. Moreover, one can change the constraint AtMost$k$Diff itself. For instance, suppose that we are only interested in the robustness of the overall makespan, and we are solving a sequence of *deadline job shop*. One can extend the deadline of P' by a given (acceptable) quantity q, and omit the AtMost$k$Diff constraint. The resulting solution will be one such that no "break" of size less than or equal to a can result in a makespan increase of more than q.

Another concern is that the semantic of the super solution depends on the model chosen for solving a problem. For instance, an efficient way of solving the job shop scheduling problem is to search over sequences of activities on each resource, rather than assigning start times to activities. In this case, two solutions involving different start times may map to a single sequence. Therefore the semantic of a super solution is changed, a break or a repair implies a modification of the order of activities for a resource. It is therefore interesting to think of ways of solving a problem using a model whilst ensuring that the solution is a super solution for another model. If it is possible to channel both representations, then one can solve one model whilst applying the `reparability` procedure to the second model.

Optimization

Finding super solutions is still, and will certainly remain a difficult problem. One way to avoid this difficulty is to try to get a solution as close as possible to a super solution. We can then start from an initial solution found using the best available method, and then we improve its *reparability* with a branch and bound algorithm.

The (a, b)-*reparability* of a solution is defined as the number of combinations of less than a variables that are covered by a b-repair. In (EHW04a), the authors report that turning the procedure into a branch and bound algorithm that maximize reparability is the most promising way of using super solutions in practice as an (a, b)-super solution may not always exist. Moreover doing so, one can get a "regular" solution with the fastest available method, and improve its reparability afterward.

The algorithm introduced here can easily be adapted in this way. The procedure `reparability` would return the number of b-repairs founds, instead of failing when a break does not accept one. The main backtracker would then backtrack either if the problem is made arc inconsistent or the value returned by `reparability` is less than the reparability of the best full solution found so far. We rewrite the procedure `reparability` adapted to this purpose in Algorithm 5.

Algorithm 5: `reparability`$(P, S, Past, a, b)$:Int

```
covered ← ∅;
foreach A ⊆ Past' such that |A| = a and A ∉ covered and Γ_{b+1} ⊈ Past' do
    foreach X_i ∈ A do
        D'(X'_i) ← D'(X'_i) − {S[i]};
    foreach X_i ∈ (X \ Γ_b(A)) do
        D'(X'_i) ← D(X_i);
    S' ← solve(P'|_{Past'} + AtMost(|A| + b)Diff(Past', S));
    if S' ≠ nil then
        Diff ← {i | S'[i] ≠ S[i]};
        foreach A' ⊆ Diff such that |A'| = a do
            covered ← covered ∪ {A};
return |covered|;
```

Unfortunately, if all other improvements still hold, we cannot prune P as a result of a pruning when pre-processing P', since a break without repair is allowed.

Future work

Our next priority is to implement this algorithm for finding super solutions to the job-shop scheduling problem. We will use a constraint solver that implements shaving, the Orr heuristic described in (SF96), as well as constant time propagators for enforcing GAC on precedence and overlapping constraints. As these two constraints are binary, the neighborhood of a variable is limited. Hence the theoretical results introduced here should apply. Moreover, we expect this reasoning to offer a good synergy with a strong global consistency method such as shaving. Indeed more inference can be done on the repairs (without searching) than with GAC, and thus more values can be pruned in the main search tree because of the robustness requirement. We therefore expect to be able to solve much larger problems than the instances solved in (EHW04a).

Conclusion

We have introduced a new algorithm for finding super solutions that improves upon the previous method. The new algorithm is more space efficient as it only requires to double the size of the original constraint satisfaction problem. The new algorithm also permits us to use any constraint toolkit to

solve the master problem as well as the sub-problems generated during search. We also take advantage of repair multidirectionality and of inference based on just a restricted *neighborhood* of constraints. Moreover, this algorithm allows the user to easily specify extra constraints on the repairs. For example, we can easily specify that all repairs should be later in time than the first break.

Acknowledgements

Emmanuel Hebrard and Toby Walsh are supported by National ICT Australia. Brahim Hnich is currently supported by Science Foundation Ireland under Grant No. 00/PI.1/C075. We also thank ILOG for a software grant.

References

J.C. Beck A.J. Davenport, C. Gefflot. Slack-based techniques for robust schedules. In *Proceedings ECP'01*, 2001.

J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *European journal of Operational Research*, 78:146–161, 1994.

B. Hnich E. Hebrard and T. Walsh. Robust solutions for constraint satisfaction and optimization. In *Proceedings ECAI'04*, 2004.

B. Hnich E. Hebrard and T. Walsh. Super solutions in constraint programming. In *Proceedings CP-AI-OR'04*, 2004.

J. Lang H. Fargier. Uncertainty in constraint satisfaction problems: a probabilistic approach. In *Proceedings ECSQARU'93*, 1993.

D. Knuth. The art of computer programming: Prefascicle 3a: Generating all combinations. http://www-csfaculty.stanford.edu/ knuth/fasc3a.ps.gz.

A. Parkes M. Ginsberg and A. Roy. Supermodels and robustness. In *Proceedings AAAI-98*, pages 334–339, 1998.

A. Cesta N. Policella, S.F. Smith and A. Oddi. Generating robust schedules through temporal flexibility. In *Proceedings ICAPS'04*, 2004.

N. Sadeh and M.S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86(1):1–41, September 1996.

R. Weigel and C. Bliek. On reformulation of constraint satisfaction problems. In *Proceedings ECAI-98*, pages 254–258, 1998.