Using the Breakout Algorithm to Identify Hard and Unsolvable Subproblems

Carlos Eisenberg and Boi Faltings

Artificial Intelligence Laboratory (LIA)
Swiss Federal Institute of Technology (EPFL)
IN-Ecublens, 1015 Lausanne, Switzerland
{eisenberg, faltings}@lia.di.epfl.ch
http://liawww.epfl.ch/

Abstract. Local search algorithms have been very successful for solving constraint satisfaction problems (CSP). However, a major weakness has been that local search is unable to detect unsolvability and is thus not suitable for highly constrained or overconstrained problems. In this paper, we present a scheme where a local search algorithm, the breakout algorithm, is used to identify hard or unsolvable subproblems. This is used in two ways.

The first to generate a fail-first variable order for a systematic backtrack search that proves unsolvability or solves the problem efficiently. The combination of the two methods is a complete algorithm. On randomly generated coloring problems, the method performs extremely well, in particular, for tightly and overconstrained CSPs.

The second way of using the breakout algorithm is as a filter for identifying possibly unsolvable subproblems. We present an efficient algorithm that guarantees to find the smallest unsolvable subproblem by systematic search.

The presented scheme is of great practical use as ideal failure analysis tool, which also supports the repair of a problem.

1 Introduction

The breakout algorithm is an efficient, local search algorithm for solving Constraint Satisfaction Problems (CSPs). The roots of the algorithm go back to Minton et al. ([6]) and Morris ([7]). Minton et.al., presents a local search algorithm, called min-conflict heuristic, which iteratively repairs a given assignment in order to minimize the number of conflicts (constraint violations). One major drawback of the min-conflict heuristic however is the possibility of being caught in a local, non solution minimum, which forces the algorithm to restart from a new initial assignment. Morris eliminated this drawback by extending the min-conflict heuristic with a breakout method that allows the search process to escape from local, non solution-minima.

The strengths of the breakout algorithm are simplicity, robustness, low memory requirement and high efficiency for solving underconstrained problems. These

properties are extremely useful when dealing with large scale constraint satisfaction problems. Minton et. al. demonstrated this by solving large-scale scheduling problems, where the method performs orders of magnitude better than traditional backtracking techniques. The major weak point of the breakout algorithm and this applies to local search in general, is its incompleteness; it cannot guarantee termination, even if a solution exists, and it will not terminate if no solution exists. In this paper we present a hybrid algorithm where we combine an incomplete, local search algorithm, the breakout algorithm, with a systematic, complete search algorithm, backtracking. By combing the breakout algorithm with backtracking, we compensate its weaknesses: incompleteness and difficulty to deal with tightly- and overconstrained problems. Moreover, we discover that the combination of the two algorithms leads to synergies. By using the weight information that is generated during the local search process, we can locate and order particularly hard or unsolvable subproblems. These can guide the complete search process such that variables of the hardest subproblems come first, providing a powerful fail-first heuristic for systematic search. Moreover, we show that by satisfying a weight sum constraint and using the graph structure, the smallest unsolvable subproblem can be efficiently identified. This result is useful for generating explanations and relaxing overconstrained problems.

The major contributions of this paper are the following results:

- an identification scheme for hard and unsolvable subproblems using the constraint weight information of the breakout algorithm
- a separation of hard and unsolvable subproblems of different sizes
- a fail-fast variable ordering heuristic, based on constraint weight
- a hybrid and complete solving algorithm for CSP's combining local search and complete search(BOBT)
- an algorithm for identifying a smallest unsolvable subproblem (BOBT-SUSP)

The rest of the paper is organized as follows. In Section 2 we give definitions, discuss the properties of unsolvable subproblems and give a brief overview of the execution of the breakout algorithm. In Section 3 we present a solving scheme and propose two new hybrid algorithms:

- BOBT (Breakout with Backtracking) as a complete mixed algorithm for solving CSPs or identifying an unsolvable subproblem if it exists, and
- BOBT-SUSP (BOBT for a smallest unsolvable sub problems) for identifying unsolvable subproblems.

In Section 4, we show the results of the experiments where we applied the BOBT algorithm to solve randomly generated graph 3-colouring problems. In Section 5, we survey related work.

2   Preliminaries

2.1   Definitions

Definition 1 (Constraint Satisfaction Problem P). *A finite, binary constraint satisfaction problem is a tuple $P = <X, D, C>$ where:*

- $X = \{x_1, .., x_n\}$ *is a set of n variables,*
- $D = \{d_1(x_1), .., d_n(x_n)\}$ *is a set of n domains, and*
- $C = \{c_1, .., c_p\}$ *is a set of p constraints, where each constraint $c_l(x_i, x_j)$ involves two variables x_i and x_j and is a function from the Cartesian product $d_i(x_i) \times d_j(x_j)$ to $\{0, 1\}$ that returns 0 whenever the value combination for x_i and x_j is allowed, and 1 otherwise (note that this is the formulation of weighted CSP). We call the set $\{x_i, x_j\}$ vars(c) and there is at most one constraint with the same set of variables.*

Definition 2 (Subproblem P_k). *A subproblem P_k of a problem P with k variables is defined as a tuple $P_k = < X_{P_k} \subseteq X, D_{P_k} \subseteq D, C_{P_k} \subseteq C >$ with the additional constraint that C_{P_k} contains all and only constraints between variables in X_{P_k}. We define the* size *of a subproblem $size(P_k)$ as the number of constraints $|C_{P_k}|$.*

Definition 3 (Unsolvable Subproblems). *A subproblem P_k is* unsolvable *if there is no value assignment to variables in X_{P_k} that satisfies all constraints in C_{P_k}.*

An unsolvable subproblem P_k is minimal *if it becomes solvable by removing any one of its variables.*

A minimal unsolvable subproblem P_k is a smallest *unsolvable subproblem of P, if there is not another minimal unsolvable subproblem P'_l such that $size(P'_l) < size(P_k)$.*

Definition 4 (Graph Connectivity GC.). *The connectivity of a variable is the number of constraints which refer to that variable. The connectivity of a graph GC is the average connectivity of the variables. See ([5]).*

2.2 The Standard Breakout Algorithm

The breakout algorithm ([7]) is a further development of the min-conflicts algorithm ([6]) and is the basis for our work.

```
1: function breakout(CSP, iteration − limit, breakout − limit)
2:  S ← random initial state
3:  W ← vector of all 1
4:  while S is not a solution ∧(iteration − limit > 0) ∧ (breakout − limit > 0) do
5:     if S is not a local minimum then
6:        make local change to minimize conflicts
7:        iteration-limit ← iteration-limit - 1
8:     else
9:        increase the weight of all currently violated constraints
10:       breakout-limit ← breakout-limit - 1
11: return(S, W)
```

Algorithm 1: Breakout algorithm.

Algorithm 1 shows the basic breakout algorithm. The state $S = < x_1 = v(x_1, S), ..., x_n = v(x_n, S) >$ is an assignment of values to all variables of the problem. It can be a solution when no constraint is violated, otherwise it has a number of conflicts with constraints. The breakout algorithm contains two essential steps: determining the local change that minimizes conflicts, and increasing the weights (called the breakout).

With every constraint, we associate a weight:

Definition 5 (Constraint Weight w). *Each constraint is assigned a weight $w(c(x_i, x_j))$ or in short $w_{i,j}$. All weights are positive integer numbers and are set to 1 initially. The breakout algorithm uses the weights in order to escape from local non- solution minima.*

In Algorithm 1, the weights are grouped together in the weight vector W.

Conflict minimization consists of choosing a variable and a new value that reduces as much as possible the conflicts in the current state. For this, we compute for every variable its conflict value, defined as follows:

Definition 6 (Variable Conflict Value ω). *The conflict value $\omega(x_i, v_a, S)$ of variable x_i assigned the value v_a in state S, is the sum of weights of the constraints involving x_i that would be violated in a state S' that differs from S only in that $x_i = v_a$:*

$$\omega(x_i, v_a, S) = \sum_{c_l(x_i, x_j)} w(c_l) \cdot c_l(x_i = v_a, x_j = v(x_j, S))$$

where $v(x_j, S)$ is the value assigned to variable x_j in state S.

The best improvement is to the variable/value combination x_i, v_a such that $\omega(x_i, v(x_i, S), S) - \omega(x_i, v_a, S)$ is largest. If there is such a combination with an improvement greater than 0, the variable/value combination with the best improvement is chosen as the local improvement.

If no improvement is possible, the algorithm is in a local minimum. In this case, the algorithm increases the weight of each violated constraint by 1, and again attempts to compute the possible improvements. Increasing the weights of each violated constraint is what we term a *breakout step*. Since the current violations will gain more weight, eventually an improvement in the conflict value will be possible; this is called the breakout.

In general, one imposes a runtime limit on the algorithm: there can be a limit on the number of *iterations*, i.e. the number of times variables are revised, or on the number of *breakout steps*.

For the breakout algorithm, we can observe the following:

Lemma 1. *After m breakout iterations, the sum of the constraint weights $w_{sum} = \sum_{c(x_i, x_j) \in C_{P_k}} w_{i,j}$ of an unsolvable subproblem P_k with $|C_{P_k}| = q$ constraints must be greater than or equal to $m + q$.*

Proof. If a subproblem is unsolvable, then in every breakout step, one or more of the subproblem constraints must be violated and the corresponding constraint

weight is increased. The lower bound for w_{sum} can be derived by assuming that in every iteration only one constraint is violated, in this case the weight sum must be equal to $m + q$.

Based on Lemma 1, we define:

Definition 7 (Weight sum condition for subproblem P_k). *We say that a subproblem P_k satisfies the weight sum condition if and only if after m iterations of the breakout algorithm, the condition of Lemma 1:*

$$\sum_{i=1}^{q} w(c_i) \geq m + q$$

is satisfied, where $c_i = c(x_s, x_t)$ are all the constraints of the constraint set C_{P_k} of the subproblem P_k, and $q = |C_{P_k}|$.

The weight sum condition is a powerful tool for searching unsolvable subproblems since by Lemma 1, any unsolvable subproblem must satisfy it:

Lemma 2. *After m iterations of the breakout algorithm, an unsolvable subproblem with q constraints must satisfy the weight sum condition.*

Proof. The condition is ensured by Lemma 1.

Thus, if after m iterations the breakout algorithm has not found a solution, and we suspect that the problem contains an unsolvable subproblem with 3 constraints, then we only have to consider subproblems whose weight sum is at least $m + 3$. If we apply this constraint in the problem of Figure 1, we find that the constraints of w1, w9, w10, whose sum is 103, are the only three constraints that satisfy the sum constraint and indeed describe an unsolvable subproblem of size 3, colouring a graph of 3 nodes with only 2 colours. Thus, the weight sum constraint is of great use for pruning the search for potential unsolvable subproblems for the algorithm BOBT-SUSP described in section 3.

When applying the breakout algorithm to small problems that are entirely unsolvable, the condition can be already applied after a small number of breakout iterations. When unsolvable subproblems are embedded in a larger structure, as shown in Figure 1, there will also be many subproblems that satisfy the weight sum condition by accident. In this case, we may need to run the breakout algorithm for a certain minimum number of cycles before the unsolvable subproblem can be reliably identified.

Considering a randomly chosen individual constraint c, we can measure the probability that after m breakout steps c is violated in a breakout step as:
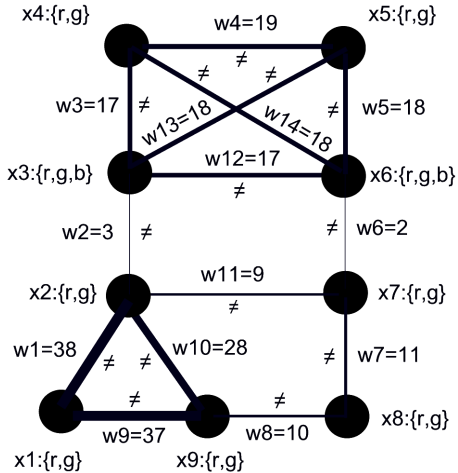
$$p(c = violated) = \frac{\sum_{c_l \in C} w(c_l) - 1}{m|C|} \tag{1}$$

When solving the problem, this probability will decrease during the first BO steps, since BO progressively eliminates conflicts. If the problem is solvable for the BO, then the probability eventually becomes 0. Otherwise, it will stabilize
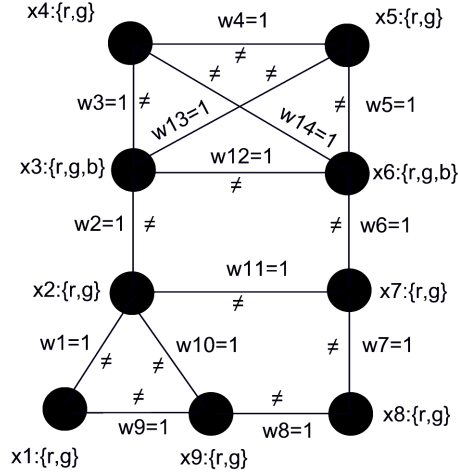
Fig. 1. The weight graph of an unsolvable graph colouring problem containing three unsolvable subproblems of size 3 (x_1, x_2, x_9), 4 (x_3, x_4, x_5, x_6) and 5 $(x_1, x_2, x_7, x_8, x_9)$, after 0 and 100 breakout steps.

and converge towards a constant value. If this is the case and BO cannot solve the problem due to a hard or unsolvable subproblem P of size q, the constraints that belong to the unsolvable subproblem are identified by the fact that their probability of being violated is at least equal to $1/q$. Thus, the expected difference in weight between a constraint that is in the unsolvable subproblem and one that is not is[1]:

$$\delta = (1/q - p(c = violated)) \cdot m \qquad (2)$$

As constraints belonging to the unsolvable subproblem can be identified only when their weights differ from the others by at least 1, we propose as a reasonable heuristic for choosing the number of breakout iterations for identifying subproblems of size q as:

$$m(q) \geq \frac{1}{1/q - p(c = violated)} \qquad (3)$$

which means that the expected difference in weight is at least 1. When higher accuracy is desired, we can of course choose a higher expected weight difference and thus a larger number of iterations.

For example, in Figure 1, the total weight of the 14 constraints after $m = 100$ breakout iterations is 245, so that the probability:

$$p(c = violated) = 231/1400 \simeq 0.165$$

---

[1] This ignores the fact that the constraints in the unsolvable subproblem itself increase the probability of constraint violations, so it is overly pessimistic.

Thus, in this problem we could identify a subproblem of size 3 after approximately $1/(1/3 - 0.165) = 1/0.16833 \leq 6$ iterations, while for a subproblem of size 5 we would need about 30 iterations, and a subproblem of size 7 could not be identified with this reliability at all since $p(c = violated)$ is larger than $1/7$.

Note that due to equation 3 this method will work very well when p(c=violated) and q are small. In this case, the minimum number of required iterations m becomes small. This means that it is always easier to identify an unsolvable subproblem of size q than a larger one of size $q' > q$. Also, it is always easier to identify an unsolvable subproblem when the average constraint violations in a breakout step is small. These conditions are not unrealistic. In practice, problems are formulated rationally and are usually not excessively overconstrained. Often they contain only a few flaws of small size and are almost feasible. With our method such flaws are easily identified and help to repair the problem. Thus, this method is particularly well suited to deal with situations where there are small unsolvable subproblems.

3 The Scheme

These observations and properties of the breakout algorithm inspired us to use the constraint weight information which is generated by the breakout algorithm for localizing the critical problem variables and thus hard or unsolvable subproblems. This idea is based on the observation that the constraint weights are also violation counters, which are incremented whenever the search is in a local minimum. Increasing the weights only in local minimum states is an advantage since in this state the noise level, generated by constraints not belonging to a hard or unsolvable subproblem is the lowest. We are now going to present a hybrid scheme where we first apply the breakout algorithm, and then switch to systematic backtrack search when no solution has been found after a given iteration limit.

When the local search method does not find a solution, we terminate and sort the variables according to the constraint weights and the graph structure. Intuitively, variables which cause the greatest conflict and thus describe the hardest part of the problem will therefore be located at the top of the ordered variable list. The subsequent complete search method will then consider those first. Beginning with the hardest part of the problem is a great advantage for systematic search algorithms. Firstly, because systematic search methods are efficient for solving highly- and over-constrained problems. Secondly, the chance of finding an unsolvable subproblem in the most constrained part of the problem is much greater than to find it in a less constrained part. For this reason, many complete search algorithms use variable ordering heuristics that order variables according to their constrainedness.

Another aspect that we are able to cover with this scheme is to give an explanation why the search for a solution fails in the form of a smallest unsolvable subproblem. This information is of great practical use because it can be exploited to repair a problem or it can be the basis for an interactive failure analysis tool.

The weight-sum constraint can be used as a highly effective filter for searching for a smallest unsolvable subproblem.

We now present algorithms for two different purposes. The first algorithm BOBT is designed to solve a CSP by a hybrid scheme of breakout algorithm and backtracking. The second algorithm BOBT-SUSP is based on the first algorithm and is extended to identify a smallest unsolvable subproblem.

3.1 Algorithm 1: Hybrid Solver BOBT

The first version of the hybrid algorithm, Algorithm 2, begins by searching for a solution using the standard breakout method. If after a bounded number of breakout iterations, the local search process has not found a solution, the process is aborted and the constraints are sorted according to their weights. Constraints with a high weight are most likely to belong to an unsolvable subproblem. Therefore, the constraint with the highest weight is selected and its variables make up the first candidate subproblem P.

The algorithm then iterates the following steps. First, it attempts to solve the subproblem P by a systematic backtrack search. If the search finds a solution, then either it has found a solution to the original problem and returns it, or the subproblem is extended by the variable x_i such that the sum of the weights of all constraints connecting x_i to P is highest. If not, then the algorithm has found an unsolvable subproblem, calls the function $musp$ to determine its minimal version (see below), and returns.

1: function BOBT$(X, D, C, maxbreak)$
2: $(S, W) \leftarrow breakout(< X, D, C >, \infty, maxbreak)$
3: **if** S is a solution **then**
4:     return(solvable, S)
5: **else**
6:     $P \leftarrow vars(argmax_{c \in C}(w(c))$
7:     **loop**
8:         $S \leftarrow backtrack - search(P, D, C)$
9:         **if** S is a solution **then**
10:            **if** $S = X$ **then**
11:                return(solvable, S)
12:            **else**
13:                $P \leftarrow P \cup \{argmax_{x_i \in X \setminus P} \sum_{c(x_i, x_j), x_j \in P} w(c)\}$
14:         **else**
15:            $musp \leftarrow musp(P, D, C)$
16:            return (unsolvable, $musp$)

Algorithm 2: Hybrid solver BOBT: returns either a solution or a minimal unsolvable subproblem.

We can show the following:

Theorem 1. *Algorithm 2 is complete: if there is a solution to the CSP, it finds it.*

Proof. If there is a solution, either it will be found by the breakout algorithm, or by backtrack search once the problem P has been extended to cover the entire problem.

Function musp is given in Algorithm 3 and is derived from the fo-search algorithm described in [3]. It is based on the following observation: assume that a backtrack search method fails and the first variable for which no assignment was found is x_i. We call x_i the *failed variable*. Then x_i is part of every unsolvable subproblem involving those variables, and thus the minimal unsolvable subproblem. The algorithm iteratively places the failed variable at the head of the variable order, and re-initiates a backtrack search. Once it reaches again a variable that had already been a failed variable before, the variables up to this variable must make up the minimal unsolvable subproblem.

1: Function musp(X,D,C)
2: $e_1, .. e_k \leftarrow unmarked$
3: **loop**
4:    $i \leftarrow 1, k \leftarrow 1$
5:    **repeat** {backtrack search}
6:      **if** $exhausted(d_i)$ **then** {backtrack}
7:        $reset - values(d_i),\ i \leftarrow i - 1$
8:      **else**
9:        $k \leftarrow max(k, i),\ x_i \leftarrow nextvalue(d_i)$
10:        **if** $consistent(\{x_1, ..., x_i\}, C)$ **then** {extend assignment}
11:          $i \leftarrow i + 1$
12:    **until** $i = 0$
13:    **if** $e_k = marked$ **then**
14:      **return** $(x_1, .., x_k)$ as the minimal usp
15:    **else**
16:      $e_k \leftarrow marked$
17:      reorder variables in X so that x_k becomes x_1

Algorithm 3: Function *musp* for extracting the minimal unsolvable subproblem.

3.2 Algorithm 2: Hybrid Solver BOBT-SUSP for identifying a smallest unsolvable subproblem

The second version of the hybrid algorithm is designed to identify a smallest unsolvable subproblem, which can be of great practical value for failure analysis and problem repair. The algorithm takes as arguments the CSP $< X, D, C >$, the weights W generated by the breakout algorithm, the number of breakout iterations m and the maximum number of constraints for the subproblem $maxsize$ to limit the search.

The algorithm systematically generates all subproblems of 2 and more variables (single constraints); it is optimized by observing that the subproblems must contain at least one constraint with weight $1 + m/maxsize$. The algorithm systematically generates all subproblems of increasing size. Applying the weight-sum criterion as a filter, it then tests all potential unsolvable subproblems for actual insolvability using backtrack search. If it finds an unsolvable problem, then it is automatically guaranteed to be the smallest, since problems were generated in increasing size. Therefore, no call to Function musp is required here.

1: Function BOBT-SUSP(X,D,C,W,m,maxsize)
2: $OPEN \leftarrow \{\{c\}|c \in C \wedge w(c) \geq 1 + m/maxsize\}$
3: $CLOSED \leftarrow \{\}$
4: **while** $OPEN \neq \{\}$ **do**
5:    $cand \leftarrow first(OPEN)$
6:    $OPEN \leftarrow rest(OPEN)$; $CLOSED \leftarrow CLOSED \cup \{cand\}$
7:    **if** $\sum_{c \in cand} w(c) \geq m + |c|$ **then**
8:      $S \leftarrow backtrack - search(cand, D, C)$
9:      **if** S is not a solution **then**
10:        **return** (cand)
11:    **if** $|cand| < max - size$ **then**
12:      **for** $x_i \in X \backslash \bigcup_{c \in cand} vars(c)$ **do**
13:        $nc \leftarrow set of constraints connecting x_i to variables in cand.$
14:        $s \leftarrow cand \cup nc$
15:        **if** $s \notin OPEN \wedge s \notin CLOSED \wedge s \neq cand$ **then**
16:          insert s into the list $OPEN$ so that $OPEN$ is ordered by the size of the subproblems.
17: **return** fail

Algorithm 4: BOBT-SUSP: an algorithm that searches for an unsolvable subproblem up to size k.

Lemma 3. *Algorithm 4 is complete in that if there is an unsolvable subproblem with less than maxsize constraints, it will find it, and sound in that the subproblem it finds is also a smallest.*

Proof. The algorithm systematically checks all subproblems in increasing order of size, so if it finds an unsolvable one, it will be the smallest. At the same time, it examines all potentially unsolvable subproblems, so it is also complete.

### 3.3 Determining the Right Breakout Iteration Bound

As discussed earlier, the capability of the method to detect unsolvable subproblems depends crucially on the iteration bound given to the breakout algorithm. Excessive breakout iterations are computationally expensive, but on the other hand we do not want to miss the unsolvable subproblems.

A pragmatic solution is to search for an unsolvable subproblem of size q using Algorithm 4 as soon as it becomes feasible according to the considerations in section 2. If no unsolvable subproblem of a size up to q_{max} has been found, Algorithm 2 should be used, as it is likely that the hardest subproblem is either solvable or very large.

For the experiments described in the next section we used a fixed breakout iteration bound value of 30 and then always switch to Algorithm 2. It is likely that by first checking for small unsolvable subproblems, better results can be obtained.

4 Experiments and Results

In order to test our presented scheme we generated a large set of 10,000 random graph 3-colouring problems according to the method described in [Davenport et.al. 1995]. The problem graphs that we generate consist of 30 variables with a connectivity of 2-6. The ratio of the solvable to the unsolvable problems is 1:1. Figures 2 and 3 show the results of the experiments as diagram. In Figure 2, we draw the number of constraint checks for BO and BOBT as function of the problem connectivity.
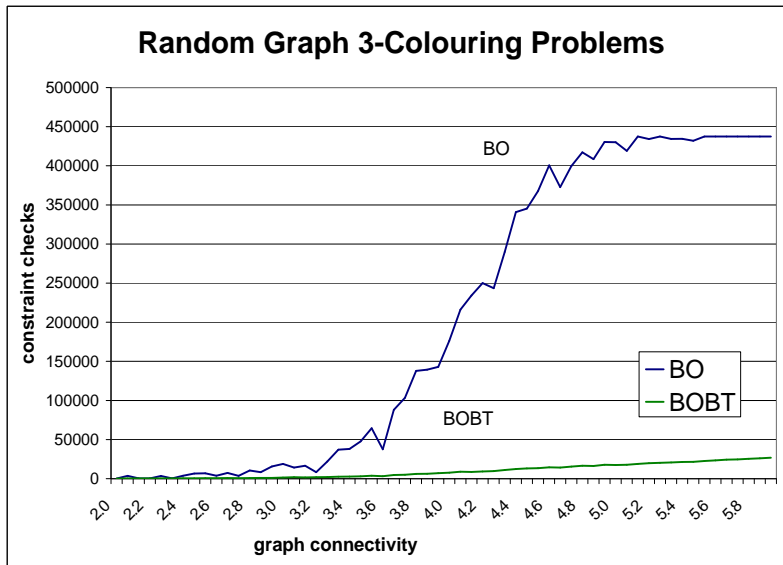
Fig. 2. Solving 10,000 randomly generated, 30 node graph 3-colouring problems, with the breakout algorithm BO and the hybrid algorithm BOBT, combining the breakout algorithm with backtracking.
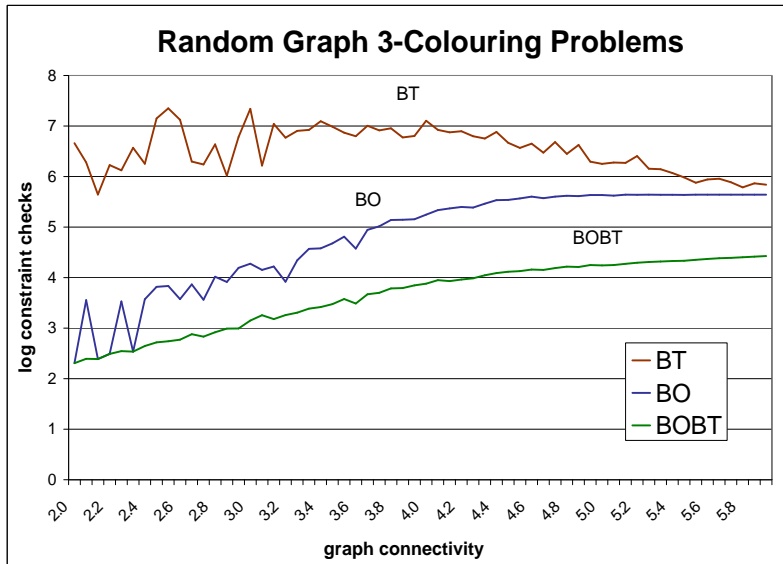
Fig. 3. Number of constraint checks on a logarithmic scale for solving 10,000 randomly generated, 30 node graph 3-colouring problems with BT, BO and BOBT.

In Figure 3, we also draw the number of constraint checks, but on a logarithmic scale. In Figure 2, we see that the breakout algorithm performs well for under constrained problems with a connectivity of 2-4. However, it lacks performance for tightly- and over-constrained problems with connectivity > 4. This result is not surprising. BO is known to perform badly for tightly constrained problems and does not terminate when the problems are unsolvable. We therefore use the bound on the number of iterations to terminate the algorithm in that case. We set this bound to $4.37 \cdot 10^5$. We chose this value since in our experiments, BO found no more solutions on tightly-constrained and over-constrained problems, so it is the fairest bound that can be set to limit useless iterations.

Looking at both figures, we observe that the hybrid algorithm BOBT clearly outperforms BT and BO for all connectivity values. This result proves the correctness of our scheme and shows the synergies of combining local search with complete search. The hybrid algorithm performs much better than the methods in isolation. Analyzing the execution of the hybrid algorithm, we notice that BO finds the most solutions for underconstrained problems, while for tightly constrained problems BT finds more solutions. We also observe, that although the backtrack search trace shows exceptionally hard problems (see [2]) in the connectivity area of 2.5-3, that no exceptionally hard problems occur when BT is used in combination with BO. The BOBT curve is much smoother than the curve of BO and BT. This result is surprising and needs further investigation. However, we already conclude that this phenomenon can be explained by the

constraint weight based variable order that BO delivers and that seems to be optimal. This conclusion is also supported by another observation. When we implement forward checking into BOBT, the average partial solution size (6.4) when the algorithm determines failure, is smaller than the size (7.2) of the currently best heuristic for graph colouring problems, the Brelaz [1] heuristic.

5   Related Work

Limited discrepancy search ([10, 11]) shares with the backtracking part of our method the idea of starting with a good initial assignment to variables and only incrementally varying it.

Pesant and Gendreau ([12]) describe a method that uses systematic search to determine the best local changes to be applied in a local search algorithm. Our ideas could be applied in a similar way by isolating subproblems that the breakout algorithm cannot solve, and feeding their solution (if any) back into the breakout algorithm as a better local move.

Similar to this approach, Shaw ([13]) proposes a method called Large Neighborhood Search that performs local search where each move consists of recomputing part of the solution using a systematic search algorithm (limited discrepancy search). The method has other features specific to vehicle routing problems that does not allow direct comparison to our method.

[14] describes a method for solving CSP using a cooperation of systematic and local search. Solvers exchange hints consisting of partial solutions to improve each other's performance, and small performance gains are shown.

Zhang and Zhang ([15]) propose to solve satisfiability problems using a method where a partial consistent assignment is generated using local search, and then extended to a solution using systematic search. The difference to this method is that we use additional information obtained from the breakout algorithm to isolate the hardest subproblems and order variables accordingly. This direction brings us larger performance gains than the undirected methods they proposed.

El Sakkout and Wallace ([16]) develop a hybrid solver called probe backtracking. This algorithm goes back to [17] and is an extended form of backtracking where the backtrack search is supported by a local search algorithm called the probe. The probe functions as lookahead procedure and directs the backtrack search towards violated regions of the probe search space with the goal to solve the harder parts of the problem first.

The difference to this method is that we essentially integrate constraint violations over a sequence of iterations and express these by the weights. The weight information then enables us to identify potential unsolvable subproblems, which we then feed into a backtracking algorithm in order to prove their unsolvability. Probe backtracking in comparison, is a difference method. The backtrack search is guided by instantaneous constraint violation snapshots, which is more sensitive to noise.

Furthermore, both methods come from a different perspective. In our scheme we are pessimistic about finding a solution and focus on identifying unsolvable subproblems. In contrast probe backtracking is optimistic about finding a solution and attempts to solve the hardest part of the problem by backtrack search and the easy part by local search.

6   Conclusion and Future Work

We have presented a scheme where the constraint weights assigned by the breakout algorithm are used to identify hard or unsolvable subproblems of a CSP. We have shown how this information can be used to identify a very efficient fail-first variable ordering, and thus to combine the breakout algorithm with backtrack search for a highly efficient overall CSP search algorithm. We have proven its performance on random constraint satisfaction problems.

We have also shown how the same method can be used to find the smallest unsolvable subproblem and thus provide distinct explanations for unsolvability of a CSP.

Local search algorithms have been very attractive since they can often find solutions to underconstrained problems very quickly. However, their applicability to more tightly constrained problems has been limited by their incompleteness. The first significant contribution of this paper is the presented general scheme that combines the breakout algorithm with a systematic search method and results in a new, hybrid algorithm. In our results we prove that this new algorithm is not only complete but it also performs extremely well and outperforms the two algorithms in isolation by several magnitudes. We are convinced that this scheme is also very well suited for solving distributed constraint satisfaction problems (DisCSP). Existing complete DisCSP algorithms have great performance problems to solve large problem instances. The required message traffic is too great and the majority of the search methods are too static and get caught in dead end branches. For example problems of 30 and more variables are still a big challenge. Distributed local search algorithms, such as the distributed breakout algorithm (DisBO) (see [8]), deliver much better performance, but also lack a termination guarantee. Projecting the results we obtained in this paper we propose that the efficiency of existing DisCSP algorithms can be greatly boosted by combining DisBO with a systematic DisCSP algorithm. Our next step is to apply the scheme for DisCSP and develop a distributed version of the presented hybrid algorithm.

The other interesting issue we will tackle with our scheme in the future is to identify the ordered set of all unsolvable subproblems for further characterizing problem classes and for performing failure analysis. In particular we plan to extend the scheme by a spectral analysis that gives us the distribution of unsolvable subproblems for random graph colouring problems.

References

1. D.Brelaz. New Methods to color the vertices of a graph. Communications of the ACM, 22(4):251-256, 1979.
2. A. Davenport and E. P. K.Tsang. An empirical investigation into the exceptionally hard problems. Technical Report CSM-239, Department of Computer Science, University of Essex, U.K., 1995.
3. B. Faltings and S. Macho-Gonzalez. Open Constraint Satisfaction. Proc. of the 8th International Conference on Principles and Practice of Constraints Programming, 2002.
4. I.P. Gent, E. MacIntyre, P. Prosser, B.M. Smith, and T. Walsh. Random constraint satisfaction: Flaws and structure. Technical Report APES-08-1998, APES Research Group, 1998.
5. I.P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In Proc. AAAI-96, 1996.
6. S. Minton, M. Johnston, A. Philips and P. Laird. Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. Artificial Intelligence, Vol. 58, P. 161-205, 1992.
7. P. Morris. The breakout method for escaping from local minima. Proc. of the 11th National Conf. on Artificial Intelligence (Washington, DC), 1993, pp. 40–45.
8. M. Yokoo and K. Hirayama. Distributed Breakout Algorithm for Solving Distributed Constraint Satisfaction Problems. Proc. of the Second International Conference on Multiagent Systems, 1996.
9. W.Zhang and L. Wittenburg. Distributed breakout revisited. In Proc. 18-th National Conf. on Artificial Intelligence (AAAI-2002), Edmonton, Canada, 2002, pp.352-357.
10. W.D. Harvey and M.L. Ginsberg: Limited Discrepancy Search, Proceedings of IJCAI-95, pp. 607-613, 1995
11. R.E. Korf: Improved Limited Discrepancy Search, Proceedings of the 13th AAAI, pp. 286-291, 1996
12. G. Pesant and M. Gendreau: A View of Local Search in Constraint Programming, Principles and Practice of Constraint Programming: Proceedings of the Second International Conference (CP'96), Springer-Verlag Lecture Notes in Computer Science 1118, 353-366, 1996
13. P. Shaw: Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems, Principles and Practice of Constraint Programming: Proceedings of the Fourth International Conference (CP'98), Springer-Verlag Lecture Notes in Computer Science 1520, pp. 417-431
14. Hogg, T. and Williams, C.P.: Solving the Really Hard Problems with Cooperative Search, Proceedings of the 11th AAAI, pp. 231-236, 1993
15. J. Zhang and H. Zhang: Combining Local Search and Backtracking for Constraint Satisfaction, Proceedings of the 13th AAAI, pp. 369-374, 1996
16. H. El Sakkout and M. Wallace, Probe Backtrack Search for Minimal Perturbation in Dynamic Scheduling, Journal: Constraints, vol 5-4, pp. 359-388, 1999
17. J.R. Purdom and N.G. Haven, Backtracking and Probing, Indiana University, Computer Science Technical Report No. 387, 1993