

Incremental Breakout Algorithm with Variable Ordering

Carlos Eisenberg and Boi Faltings

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

Abstract

This paper presents the Incremental Breakout Algorithm with Variable Ordering (IncBA). This algorithm belongs to the class of local search algorithms for solving Constraint Satisfaction Problems (CSP), and is based on the standard breakout algorithm extended by an incremental solving scheme combined with variable ordering. By evaluating this algorithm with two large sets of Graph Colouring and Scheduling problems, we show that it outperforms the standard breakout method by requiring 10^1 - 10^3 less constraint checks for solving under- and medium-constrained problems. Since the proposed scheme is extremely simple, it can be easily applied to other local search methods. With regards to scheduling problems we show that the algorithm delivers solutions of better quality and thus has optimization properties. In this paper we formalize this algorithm, prove its correctness and describe the execution details in the form of pseudo code. At the end of the paper we summarize the results and draw our conclusions.

Introduction

The Breakout Algorithm (BA) (Morris 93) is a local search algorithm, commonly used for solving Constraint Satisfaction Problems (CSPs). The origins of the algorithm go back to (Minton et.al. 92) and (Morris 93). Minton et.al. proposed to guide local search by a min-conflict heuristic that attempts to iteratively repair a given assignment until all conflicts are eliminated. This method proved to be very successful. It was by order of magnitudes faster than traditional backtracking techniques.

However, the min-conflict search technique had two major drawbacks. Firstly, it could get stuck at local, non-solution minima. Secondly, it could not guarantee completeness. In 1993 Morris proposed the Breakout method that could escape from these local minima and eliminate the first drawback. The second drawback could not be eliminated. Although Morris could prove that an idealized version of the Breakout Algorithm was complete, in practice the algorithm remains incomplete.

Moreover, both authors observed that the performance of the local search method was sensitive to the initial variable assignment. They observed that with an initial, conflict reduced variable value assignment, the time to reach the first local minimum, could be significantly shortened. This observation was never investigated further, and is now topic of this paper.

Another weak point of the local search methods are redundant variable revisions for under-constrained variables. Especially when problems contain a hard sub problem, and the algorithms go through many iteration, the redundant variable revisions are expensive. Backtracking based algorithms in comparison, construct solutions, and solve the problem by incrementally by adding variables with consistent values to a growing sub-problem. This technique bears two advantages. Firstly, when the partial problem is small, only a limited number of backtracks and constraint-checks are required for solving the sub-problem. Secondly, by solving the problem incrementally, variables can be ordered and permit to focus on solving the hardest sub-problem first (Gent et. al. 96).

Verfaillie et.al. 96 and Messeguer et.al. 01 have shown, that incremental problem solving can boost the search efficiency. In their work, an incremental search algorithm (Russian Doll Search algorithm - RDS) is used. This algorithm replaces one search by a number of successive searches on nested sub-problems. The optimal solutions of the sub-problems are then used as the lower bounds for solving the preceding problem.

With regards to variable ordering, many works, (see e.g. (Haralick et al. 80) and (Bachus et al. 95)), have shown, that variable ordering can boost the search performance of CSP algorithms. Kwan et.al. 95 even argue, that variable ordering has such a strong impact on search, that comparing algorithms without considering the applied variable ordering heuristics, can be misleading. However, variable ordering so far has only been considered in relation to systematic search, and never in relation to local search methods. In this paper we want to investigate, if incremental problem solving, combined with variable ordering, cannot also be applied for local search method, and improve their search efficiency.

Definitions

Definition 1: Constraint Satisfaction Problem (CSP). A complete, finite and binary constraint satisfaction problem P is defined by the triple: $P = (X, D, C)$. In this triple, $X = \{x_1, \dots, x_n\}$ is a finite set of n variables. $D = \{d(x_1), \dots, d(x_n)\}$ is a finite set of n domains for each variable in X . A domain $d(x_i)$ is a finite set of possible values for variable x_i ($1 \leq i \leq n$). $C = \{c_1(x_k, x_l), \dots, c_m(x_p, x_q)\}$ is a set of m binary constraints on an ordered pair of variables.

A solution for a CSP is a variable value assignment, so that all constraints can be satisfied simultaneously.

Definition 2: Constraint Weights. For escaping from local minima, the Breakout Algorithm assigns a weight to each constraint. A weight $w_k(c_k(x_i, x_j))$ is a positive integer, initially set to 1 and refers to the constraint $c_k(x_i, x_j)$.

Definition 3: Consistency Check. Verifying if a pair of variables (x_i, x_j) satisfy the constraint $c_{i,j}$ is called a consistency check (ccheck). The function $consistent(c_k(x_i, x_j))$ returns true (false) when the constraint is satisfied (not satisfied).

Definition 4: Variable Conflict Value. The conflict value of a variable x_i is defined as the sum of weights of the violated constraints between variable x_i and a second variable x_j .

$$variable_conflict_value(x_i) = \sum_{k=1}^m w(c_k(x_p, x_q)) \quad (1)$$

$$(x_p = x_i \vee x_q = x_i) \wedge \neg consistent(c_k(x_p, x_q))$$

Definition 5: Problem Conflict Value. The problem conflict value P , is the sum of weights of all violated constraints.

$$problem_conflict_value(P) := \sum_{k=1}^m w(c_k(x_p, x_q)) \quad (2)$$

where x_p, x_q are elements of X and all constraints $c_k(x_p, x_q)$ are inconsistent.

Definition 6: Problem Connectivity. The problem connectivity (see Gent et.al. 96), is defined by:

$$problem_connectivity := \frac{2 \cdot m}{n}, \text{ where} \quad (3)$$

m := number of constraints, and n := number of variables.

Algorithms

Breakout Algorithm (BA)

The Breakout Algorithm starts by initializing each variable $x_i \in X$ by a value, that is randomly chosen from its domain $d(x_i) \in D$. All constraint weights are initially set to 1: ($\forall w(c_k(x_i, x_j)), c_k \in C, 1 \leq k \leq m$ - number of constraints) [$w(c_k(x_i, x_j)) \leftarrow 1$]. Then the algorithm revises each variable, and labels it with the domain value that minimizes the variable conflict value. After all variables

are revised, the algorithm determines the problem conflict value. If it is 0, the algorithm terminates. If it is greater than 0, all variables are revised again. When the algorithm falls into a local, non-solution minimum, the weights of the violated constraints are increased by 1. BA repeats these steps until all conflicts are eliminated or until the maximum number of iterations is reached.

Algorithm 1: Breakout Algorithm BA	
function BA(in: X, D, C : set, in: max_iterations: integer): boolean	
1 Initialize(X, D, C);	
2 problem_conflict_value \leftarrow 1;	
3 previous_problem_conflict_value \leftarrow 1;	
4 ite \leftarrow 0;	
5 while (problem_conflict_value > 0 and ite ? max_iterations)	
6 ite \leftarrow ite + 1;	
7 for each $x_i \in X$ do	
8 if (Variable_Conflict_Value(x_i, X, C) > 0) then	
9 $x_i \leftarrow$ Revised_Value($x_i, d(x_i), X, C$);	
10 problem_conflict_value = Problem_Conflict_Value(X, C);	
11 if (problem_conflict_value = 0) then return true ;	
12 if (problem_conflict_value = previous_problem_conflict_value) then	
13 Increase_Weights(X, C);	
14 previous_problem_conflict_value = problem_conflict_value;	
15 return false ;	

Figure 1: Pseudo code of the Breakout Algorithm.

Algorithm 3: Sub-Procedures for BA and IncBA	
procedure Initialize (in: X, D, C : set)	
1 for each $x_i \in X$ do	
2 if (x_i = null) then $x_i \leftarrow$ random_value($d(x_i)$);	
3 for each $w(c_k(x_i, x_j)) \in C$ do	
4 $w(c_k(x_i, x_j)) \leftarrow$ 1;	
function Revised_Value (in: x_i :variable, in: $d(x_i)$:domain, in: X, C :set): domain value	
1 min_conflict \leftarrow Variable_Conflict_Value(x_i, X, C);	
2 $v_{min_conflict} \leftarrow$ value(x_i);	
3 for each $v_j \in d(x_i)$ do	
4 $x_i \leftarrow v_j$	
5 conflict \leftarrow Variable_Conflict_Value(x_i, X, C);	
6 if (conflict = 0) then return v_j ;	
7 if (conflict < min_conflict) then	
8 min_conflict = conflict;	
9 $v_{min_conflict} \leftarrow v_j$;	
10 return $v_{min_conflict}$;	
function Variable_Conflict_Value (in: x_i :variable, in: X, C :set): integer	
1 variable_conflict_value \leftarrow 0;	
2 for each $c_k(x_s, x_t) \in C$ and $(x_s, x_t \in X)$ and ($(x_s = x_i)$ or $(x_t = x_i)$) do	
3 if ($c_k(x_s, x_t)$ = false) then conflict_value \leftarrow conflict_value + $w(c_k(x_s, x_t))$;	
4 return variable_conflict_value;	
function Problem_Conflict_Value (in: X, C :set): integer	
1 problem_conflict_value \leftarrow 0;	
2 for each $c_k(x_s, x_t) \in C$ and $x_s, x_t \in X$ do	
3 if ($c_k(x_s, x_t)$ = false) then	
4 problem_conflict_value \leftarrow problem_conflict_value + $w(c_k(x_s, x_t))$;	
4 return problem_conflict_value;	
procedure Increase_Weights (in: X, C :set)	
1 for each $c_k(x_i, x_j) \in C$ and $x_i, x_j \in X$ do	
2 if ($c_k(x_i, x_j)$ = false) then $w(c_k(x_i, x_j)) \leftarrow w(c_k(x_i, x_j)) + 1$;	

Figure 2: Pseudo code of the sub-procedures BA & IncBA.

Incremental Breakout Algorithm (IncBA)

The Incremental Breakout Algorithm starts by setting up an empty variable set Q , $Q \leftarrow \emptyset$. Then the first variable x_i , is added to the partial variable set ($Q \leftarrow Q \cup x_i$) and IncBA assigns the value to x_i , which is returned from the *Revised_Value* function. This function searches for the value $v \in d(x_i)$, which minimizes the variable conflict value of x_i . If the variable conflict value is 0, IncBA proceeds by adding the next variable to Q . If the variable conflict value is greater than 0, IncBA proceeds by solving the sub-problem Q . In this case, IncBA revises variable by variable in Q , until the maximum number of iterations is exceeded or until the sub-problem is solved. If the maximum number of iterations is reached, the algorithm terminates and returns false. When IncBA finds a partial solution, it continues by adding the next variable: $Q \leftarrow Q \cup x_{i+1}$. When all variables are added to Q , and all conflicts are eliminated, the problem is solved and IncBA returns true.

```

Algorithm 2: Incremental Breakout Algorithm
function IncBA(in:  $X, D, C$ : set, in: max_iterations: integer): Boolean
1  $Q \leftarrow \emptyset$ , ite  $\leftarrow 0$ ;
2 while  $X \ll \emptyset$  do
3    $x_i \leftarrow \text{select\_next\_var}(X)$ ; // variable selection according to
                                     // the variable ordering heuristic
4    $X \leftarrow X / x_i$ ; // remove  $x_i$  from  $X$ 
5    $Q \leftarrow Q \cup x_i$ ;
6    $x_i \leftarrow \text{Revised\_Value}(x_i, d(x_i), Q, C)$ ;
7   if (Variable_Conflict_Value( $x_i, Q, C$ ) > 0) then
8     problem_conflict_value  $\leftarrow 1$ ; previous_problem_conflict_value  $\leftarrow 1$ ;
9     while (problem_conflict_value > 0)
10      ite  $\leftarrow \text{ite} + 1$ ;
11      if (ite > max_iterations) then return false;
12      for each  $x_i \in Q$  do
13        if (Variable_Conflict_Value( $x_i, Q, C$ ) > 0) then
14           $x_i \leftarrow \text{Revised\_Value}(x_i, d(x_i), Q, C)$ ;
15        problem_conflict_value  $\leftarrow \text{Problem\_Conflict\_Value}(Q, C)$ ;
16        if (problem_conflict_value = previous_problem_conflict_value) then
17          Increase_Weights( $Q, C$ );
18        previous_problem_conflict_value  $\leftarrow \text{problem\_conflict\_value}$ ;
19 return true;

```

Figure 3: Pseudo code of IncBA.

Algorithm Correctness

Let us briefly prove the correctness of IncBA. An algorithm is said to be correct, if for every input instance, it terminates with the correct output. When IncBA terminates with false, the output is correct because IncBA can only exit with false if it exceeded the maximum number of iterations and thus has not found a solution.

When IncBA terminates with true, a solution is found, and we have to prove that this solution is correct. IncBA can only return true, when all variables are added to Q , and if either the last variable(s) was (were) added and assigned a consistent value, or, the inner BA procedure loop (line 9-18) was left. In both cases it means the

problem conflict value was 0. This proves the algorithm correctness.

Variable Ordering

For the Incremental Breakout Algorithm we introduce dynamic variable ordering. The variable order determines the sequence, in which the variables are added to the partial variable set Q . Analogue to systematic search algorithms, the ordering of variables for local search algorithms is problem specific. For IncBA we use the proposed ordering heuristics for complete search algorithms.

- **Variable Order for Graph Colouring Problems**

For the graph-colouring problem, the fail first (FF) and the Brélaz (BZ) (Davenport 95) heuristics are the two most successful dynamic variable ordering heuristics. The fail-first (FF) heuristic, first selects the unlabeled variable with the smallest domain, which is most likely to fail next. The Brélaz (BZ) heuristic, is an extension of the FF heuristic, and first selects from the unlabelled variables with the smallest domains, the one, that is connected to the greatest number of unlabelled variables.

- **Variable Order for Scheduling Problems**

For scheduling problems we use the precedence constraint based variable ordering heuristic (PC). This heuristic first selects the tasks (variables) whose predecessors are either already labelled, or those which do not have predecessors. Predecessors of a task are all tasks, which have an outgoing precedence constraint to that task.

Results

Solving Graph Colouring Problems

The BA and three versions of the IncBA, without and with variable ordering, IncBA, IncBA-FF and IncBA-BZ, were tested on a large set of graph 3 - colourability problems (Davenport 95). For the experiments, we generated 100,000 problems with a connectivity between 2 and 5 and a step size of 0.1. We used the connectivity for evaluating the constrainedness of the problem. Each of the problems included 50 variables and the constraints were randomly generated. The phase transition of the problems occurred at a connectivity of 4.6. The algorithms were exclusively tested on soluble problems that were determined by a complete search algorithm. We counted the number of constraint checks and drew the average value as function of the graph connectivity.

Figure 4, and the appendix figures 6 - 8 show the results of the experiments. Up to a connectivity of 3.7, all incremental versions of the BA outperform the BA. Amongst the 3 incremental versions, those with variable ordering, clearly outperform the one without. Comparing the variable ordering algorithms with each other, the BZ heuristic performs better than the FF heuristic. This corresponds to the results observed for complete search

algorithms (Davenport et. al 95). Comparing the best algorithm with the worst in that region, IncBA-BZ on average requires only 8% of the constraint checks of BA.

In the connectivity region from 3.7 – 3.8 the situation changes. Here, IncBA-FF and IncBA-BZ show sudden peaks above the BA and IncBA graphs. We explain this peak by the appearance of ‘exceptionally hard problems’ (Davenport et. al 95). Although the concentration of these problems in this region was very low (< 0.001%) it caused the dramatic peaks. Surprisingly, these problems do not turn out to be exceptionally hard for BA and IncBA. Since all algorithms solved identical problems, this implies that the variable order itself is responsible for problems to become exceptionally hard.

At a connectivity of 4 until 4.5, where the phase transition occurs, the performance of all algorithms is almost equal. At a connectivity of 4.5 IncBA-BZ falls sharply and can again outperform the other algorithms by 1 magnitude.

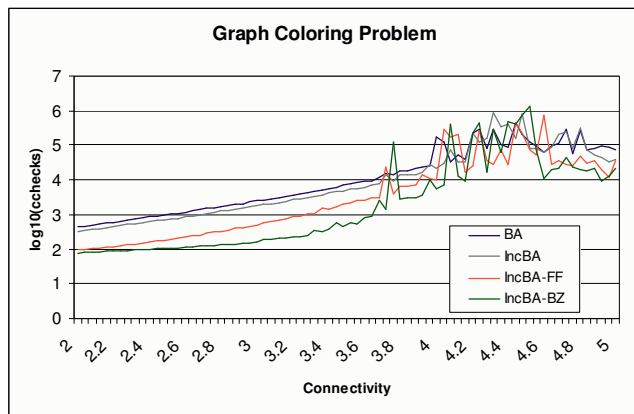


Figure 4: Graph 3 colouring problems solved with BA, IncBA, IncBA-FF and IncBA-BZ. The 4 graphs show the number of constraint checks as function of the constraint graph connectivity on a logarithmic scale.

Solving Scheduling Problems

The algorithm search performance was further evaluated by solving scheduling problems. For this experiment, three different algorithms, BA, IncBA and IncBA-PC, were developed and tested on a large set of 100,000 randomly generated problems. Each of the scheduling problems had a fixed start and finish date, consisted of 25 tasks with equal duration and included two resources; a unary and a discrete resource. For each problem between 1-25 precedence constraints, 4-14 unary resource requests and 4-25 discrete resource requests were generated and randomly distributed amongst the tasks.

The connectivity values of the generated scheduling problems ranged between 1-32. A systematic search algorithm for separating the soluble from the none-soluble problems, was not available. We therefore terminated the algorithms after $30 \cdot 10^6$ constraint checks, if no solution was obtained. Figure 5 and figures 9-11 in the appendix show the results of the experiments.

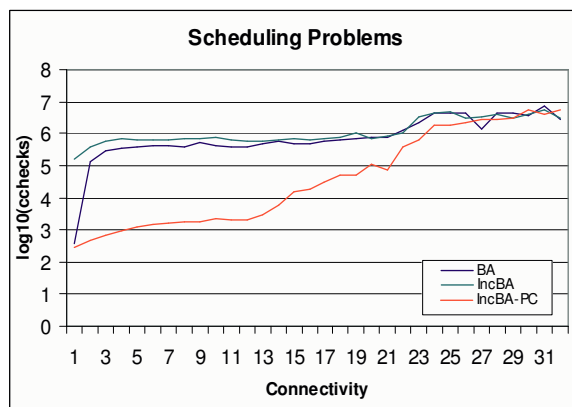


Figure 5: Scheduling Problems solved with BA, IncBA and IncBA-PC. The graphs show the number of constraint checks as function of the constraint graph connectivity on a logarithmic scale.

IncBA-PC clearly outperforms BA and IncBA. Up to a connectivity of 13, BA and IncBA need on average 10^3 more constraint checks than IncBA-PC. This is an impressive result. Only for tightly constrained problems, the number of constraint checks of IncBA-PC reaches the one of BA and IncBA. This performance boost can be explained by the PC variable ordering heuristic. With this ordering heuristic, the constraint check intensive BA subroutine (line 9-18 in the IncBA pseudo code) was rarely called. This means that for the greatest part of the unlabelled variables, IncBA-PC found a consistent labelling immediately with the *revised_value* function. By solving the largest part of the problem with the *revised_value* function, the search became extremely efficient. Only when the problems were tightly constrained, exhaustive search with the BA procedure started and caused a high number of constraint checks. The constructive variable labelling procedure implied another advantage. By assigning the earliest possible start times to tasks lead to a tighter resource utilization and thus to a better schedule quality.

Conclusion

In this paper, we have introduced a new solving scheme for local search algorithms - incremental problem solving with variable ordering. We have implemented this scheme into the Breakout Algorithm and showed that it boosts the search efficiency by 1 - 3 orders of magnitude for graph colouring and scheduling problems. The experimental results show that the incremental solving scheme alone does not lead to performance gains, but if it is combined with variable ordering, the gain is remarkable. The performance gains are enormous for under- and medium-constrained problems. For tightly constrained problems however, the performance gain is negligible. We also want to emphasize, that IncBA-PC can be considered as constraint satisfaction optimisation algorithm. It delivered schedules of better quality. Since the presented scheme is

extremely simple, we advocate its use and implementation for other local search algorithms.

References

Bacchus, F. & van Run, P. 1995. *Dynamic Variable Ordering in CSPs*. In Proceedings the First International Conference on Principle and Practice of Constraint Programming, 258-275.

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.

Dechter R. and Meiri I. *Experimental evaluation of preprocessing algorithms for constraint satisfaction problems*. Artificial Intelligence, 68:211--241, 1994.

Gent I., MacIntyre E., Prosser P., and Walsh T. "The constrainedness of search". In Proc. AAI-96, 1996.

Haralick R. and Elliott G. *Increasing tree search efficiency for constraint satisfaction problems*. Artificial Intelligence, 14:263--313, 1980.

Kwan A., Tsang E. P. K.. *Comparing CSP algorithms without considering variable ordering heuristics can be misleading*. CSM-262, Colchester, UK, 1995.

Meseguer P., Sánchez M.. *Specializing Russian Doll Search*, Lecture Notes in Computer Science, Vol.2239, p.464, 2001.

Minton S., Johnston M.D., Philips A.B. and Laird P. *Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling*. Artificial Intelligence, 58:161--205, 1992.

Morris P., *The breakout method for escaping from local minima*, Proc. of the 11th National Conf. on Artificial Intelligence (Washington, DC), 1993, pp. 40--45.

Verfaillie G., Lemaitre M. and Schiex T.. *Russian Doll Search for Solving Constraint Optimization Problems*. In Proc. of the 13th National Conference on Artificial Intelligence (AAAI-96), pages 181--187, Portland, OR, USA, 1996.

Appendix

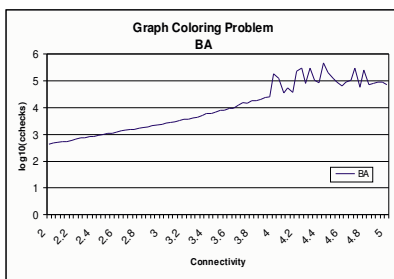


Figure 6: BA – Graph Colouring Problems.

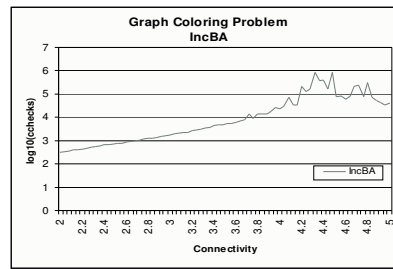


Figure 7: IncBA – Graph Colouring Problems.

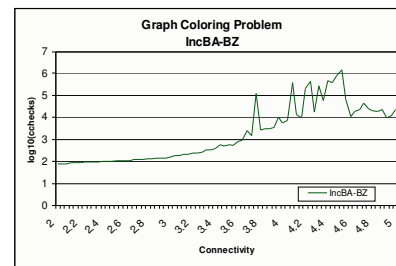


Figure 8: IncBA-BZ - Graph Colouring Problems.

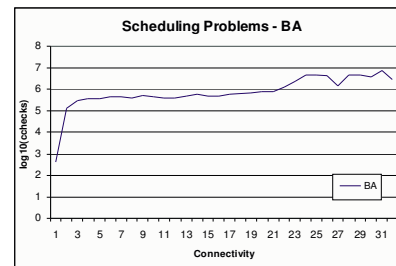


Figure 9: BA – Scheduling Problems.

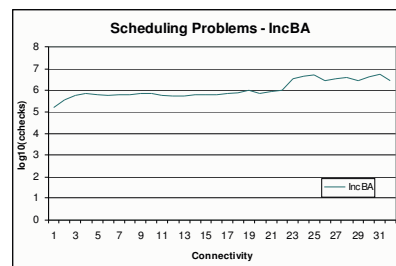


Figure 10: IncBA – Scheduling Problems.

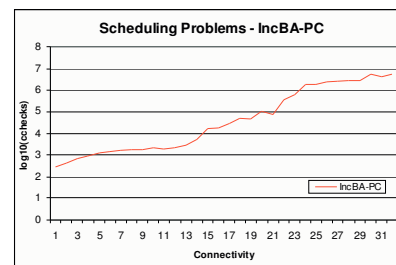


Figure 11: IncBA-PC - Scheduling Problems.