Progressive Random Broadening

Tristan Cazenave

Labo IA, Université Paris 8, 2 rue de la Liberté, 93526, St-Denis, France
cazenave@ai.univ-paris8.fr

Abstract. We describe a new algorithm that progressively broadens the number of values that can be tried for some variables in a backtrack style search. The algorithm is more progressive than similar algorithms such as iterative broadening or limited discrepancy search, and performs better than them and forward checking for the quasi-group completion problem. The algorithm uses random choices to select the variables to broaden. It is compatible with variable and value ordering heuristics, and enables randomization while retaining the performance of the heuristics. Experimental results are given for the quasi-group completion problem.

1 Introduction

The paper describes a search algorithm that progressively increases the number of variables that can be searched with a maximum broadening threshold. The variables that are allowed to increase their threshold are chosen randomly. The algorithm performs better than forward checking, iterative broadening [1] and limited discrepancy search [2] for various value ordering heuristics. The originality of the algorithm is that it randomly chooses variables to broaden, instead of randomly choosing variables to instantiate as previously tried [3, 4].

The second section presents the quasi-group completion problem, the third section details related search algorithms, the fourth section deals with the progressive random broadening algorithm, the fifth section presents experimental results, the last section concludes and outlines future work.

2 The Quasi-group Completion Problem

This section starts with describing the quasi-group completion problem, the second subsection details how problems have been generated, the third subsection explains how redundant modeling can help solving the problem.

2.1 Description of the problem

A quasi-group is an $n \times n$ multiplication table which defines a Latin square. It is a matrix such that each row and each column contains n different elements. There are only n possible values for the cells of the matrix. The size of an $n \times n$ quasi-group is n.

The quasi-group completion problem (QCP) consists in completing a quasi-group with holes (unassigned variables).

QCP is an ideal testbed for constraint satisfaction algorithms as it has structure and it is easy to generate arbitrarily hard problem instances [5]. The problem has a phase transition phenomenon which follows an easy-hard-easy pattern, depending on the number of unassigned variables. This phase transition follows the backbones of the problems: i.e. the variables that are fully constrained and take the same value in all solutions.

QCP has also been used to discover important properties of search algorithms such as heavy-tailed behavior [6] and the associated rapid randomized restarts (RRR) strategies [3, 4].

Moreover, it is related to real world applications such as statistical design, error correcting codes, conflict free wavelength routing and timetabling [7, 8].

2.2 Problem generation

In order to generate a problem, our algorithm progressive random broadening with random value ordering is used to generate a complete quasi-group of a given size. Then variables are randomly selected among the assigned variables (initially all the variables) and are unassigned until the number of holes (unassigned variables) corresponds to the desired percentage of the total number of variables. For each size, and for each percentage, 50 problems have been generated.

2.3 Redundant modeling

Redundant modeling has been used with success for QCP [7]. The primal and natural model for a QCP of order n is to take as the n^2 variables the cells of the matrix, the common initial domain for the variables being $D = \{k | 1 \leq k \leq n\}$. The variables can be named x_{ij} where i is the row and j the column of the variable. The n^2 constraints for the rows are $x_{ij} \neq x_{il}$ with $j \neq l$, and the n^2 constraints for the columns are $x_{ij} \neq x_{lj}$ with $i \neq l$. The row dual model consists in considering in which column in a given row is a given color. There are n^2 constraints of the form $r_{ik} \neq r_{il}$ with $l \neq k$, and n^2 constraints of the form $r_{ik} \neq r_{jk}$ with $i \neq j$. The column dual model consists in considering in which row in a given column is a given color. There are n^2 constraints of the form $c_{jk} \neq c_{jl}$ with $k \neq l$, and n^2 constraints of the form $c_{jk} \neq c_{lk}$ with $j \neq l$. The row channeling constraints are: $x_{ij} = k \Leftrightarrow r_{ik} = j$. The column channeling constraints are: $x_{ij} = k \Leftrightarrow c_{jk} = i$. This is exactly the redundant modeling used in [7]. A good value ordering heuristic associated to this redundant modeling is the *min-domain-sum value selection heuristic* (vdom+). It consists in choosing the value whose corresponding two variables have a minimal sum of domains sizes. It is also interesting to note that Forward Checking with redundant models is almost equivalent to Arc Consistency for QCP, except for the order of instantiation of the singleton variables.

An alternative modeling of the problem is to use $2n$ all different n-ary constraints [9].

3 Related Search Algorithms

Related algorithms are Rapid Randomized Restarts, Iterative Broadening and Limited Discrepancy Search which are presented in this order in this section.

3.1 Rapid Randomized Restarts

Rapid Randomized Restarts [3, 4] adds randomization to a constraint satisfaction algorithm by randomly choosing the next variable to assign when several variables are ranked equally, or randomly choosing a variable between variables that are ranked within H-percent of the highest scoring variable. The algorithm consists in stopping the search for a given threshold and restarting it, possibly increasing the threshold. It has been shown to eliminate the heavy-tailed behavior of backtrack style search [6].

3.2 Iterative broadening

Iterative broadening (IB) [1] consists in imposing an artificial breadth limit on the search. If the breadth cutoff is n, it stops searching at every node when n values have been tried. Iterative broadening starts searching with an artificial breadth cutoff of 2, then try with a breadth cutoff of 3, and so on, until an answer is found.

3.3 Limited discrepancy search

Limited Discrepancy Search (LDS) was proposed by Harvey and Ginsberg [2]. A discrepancy is a node in the search tree where the algorithm does not follow the heuristic. Given a leaf in a search tree, its discrepancy order is the number of discrepancies that have been necessary to reach it. LDS explores leaves in increasing discrepancy orders. Korf improved LDS with Improved Limited Discrepancy Search (ILDS) [10] which avoids revisiting leaf nodes at the depth limit with lower discrepancy orders. T. Walsh proposes. Depth-bounded limited discrepancy search (DDS) [11] which focus on branching decisions at the top of the search tree where solution are more likely to be wrong.. A closely related algorithm is Interleaved and Discrepancy Based Search [12] which has been applied with success to the quasi-group completion problem.

4 Progressive Random Broadening

Progressive random broadening (PRB) retains from iterative broadening the idea of searching trees of higher complexity at each step, enlarging the breadth cutoff of some variables at each broadening step. Unlike iterative broadening, it does not increase the cutoff by one for all the nodes, but only for a subset of them. The number of nodes that are selected for the increased breadth cutoff is limited. Let the order of a leaf for a given cutoff be the number of nodes above the leaf that are associated to the cutoff. PRB only visits the leaves that have less than a fixed order (named max_order in the algorithm below).

A PRB search depends on three parameters:

- Its *order*, which is the maximal artificial breadth cutoff allowed at any node.
- Its *nb_order_max*, which is the maximum number of nodes in a path from the root to a leaf that are allowed to have the *order* cutoff. All the other nodes on the path (i.e. at least depth of the search - *nb_order_max* nodes) have an artificial cutoff of $order - 1$.
- Its probability *proba*, which is the probability that a variable is associated to the *order* cutoff. The probability is related to *nb_order_max* as it is equal to *nb_order_max*/*max_depth*, *max_depth* being the number of variables in the problem.

Given a function $rnd(0)$ that sends back pseudo-random real numbers between 0.0 and 1.0, the core of the algorithm is as follows:

```
bool prand (order, proba, nb_order_max) {
  if (timeout ())
    return false;
  d = chooseVariable ();
  if (d == NULL)
    return true;
  if ((nb_order_max == 0) || (rnd (0) > proba))
    localorder = order - 1;
  else {
    localorder = order;
    nb_order_max--;
  }
  for (k = 0; ((k < localorder) && (k < nb_values)); k++) {
    value = chooseValue (k);
    affectValue (d, value);
    if (consistent (d, value))
      if (prand (order, proba, nb_order_max))
        return true;
    restoreVariable ();
    undoAffectValue (d, value);
  }
  return false;
}
```

In the function prand, *order* is the maximum number of values that will be considered for each variable assignment during the search. The function prand allows *nb_order_max* variable assignments with a given order, it implies that when all the variables are assigned, at most *nb_order_max* of them are being assigned with the value that comes in the *order* place in the value ordering. An important feature of the function is that the places where the variables are branched on *order* times are chosen randomly. Given the *proba* probability, each time the algorithm has to choose the number of values it will search, it chooses to search *order* values with a probability *proba*, and to search $order - 1$ values with a probability $1 - proba$, provided less than *nb_order_max* variables above the nodes are associated to the *order* cutoff, in which case, it always chooses to search only *order* values.

The prand function corresponds to one step of the PRB algorithm. A function to progressively increase the *nb_order_max* and the *order* is needed. We have tested basic functions for finding the next values of *nb_order_max* and *order*. The *nb_order_max* parameter is increased by a constant at each step, and when it becomes greater than *max_depth*, it is set back to 1, and *order* is increased by one. This is achieved with the following code:

```
int size = 30;
int inc_prand = 20;

next_nb_order_max (nb_order_max) { return nb_order_max + inc_prand; }

next_order (order) { return order + 1; }

bool prb () {
  for (order = 2; order < size; order = nextOrder (order)) {
    for (nb_order_max = 1; nb_order_max < max_depth;
         nb_order_max = next_nb_order_max (nb_order_max)) {
      proba = nb_order_max /  max_depth;
      if (prand (order, proba, nb_order_max))
        return true;
      if (timeout ())
        return false;
    }
  }
  return false;
}
```

The functions for finding the next values for *order* and *nb_order_max* can be refined to have better properties. For example the function could be designed such that the average time of *prand* with a given *order* and *nb_order_max* is twice the average cost of the previous call. If we manage to use such functions, it will ensure that the complete search for finding that the problem is not solvable is only twice the cost of the complete search without using progressive random broadening.

The current function we use are simple and can be refined. An interesting topic for further research is to estimate the balance between the search costs saved by different variation of the steps for finding the next *order* and *nb_order_max*, and the search costs incurred for proving the problem is unsolvable with such steps.

According to our experimental results, the cost of solving solvable problems decreases when we increase the step for nb_order_max from 1 to 20 (see bottom of figure 2), and the cost of finding that problems are unsolvable also decreases as we increase this step. However, there is a step after 20 when the progressive random broadening algorithm starts to increase its problem solving time (PRB with a step of max_depth is equivalent to iterative broadening which is slower than PRB with a step of 20). It would be interesting to evaluate the properties of the problem solving costs with different functions for increasing steps.

5 Experimental results

Experiments use a Celeron 1.7 GHz. In all our experiments we have used the fail first heuristic for variable selection. We select the variable that has the smallest domain size in the primal model. The default increment to nb_order_max used in PRB is set to 20 unless specified otherwise (see bottom of figure 2).

Table 1 gives the cumulative time and the number of solved problems for 1000 QCP problems of size 30, there are 50 different problems every 5%, starting at 1% of holes (50 problems with 1% of holes, 50 problems with 6% of holes, and so on until 96%). The upper part of the table gives the results using the random value ordering heuristic, while the lower part gives the results for the *vdom+* value ordering heuristic. Similar orderings between the algorithms arise with the two value ordering heuristics. Forward Checking and Iterative Broadening performs similarly in both cases. LDS performs better than FC and IB, and PRB performs better than all the others. It solves more problems in less time.

Table 1. Cumulated time for 50 QCP of size 30 every 5% with a timeout of 100 s.

Algorithm	Time	Solved
$FC(rand)$	44714	575
$IB(rand)$	44173	574
$LDS(rand)$	33986	705
$PRB(rand)$	22674	810
$FC(vdom+)$	25306	770
$IB(vdom+)$	25231	771
$LDS(vdom+)$	18736	853
$PRB(vdom+)$	11852	909

Table 2 gives similar results for QCP of size 15 with a timeout of 100 seconds. It is noticeable that the savings are much more important in table 2 than in table 1. We think this is due to the relatively low timeout of 100 seconds for problems of size 30. We suspect that if we increase the timeout for problems of size 30, the savings due to PRB will grow similarly as it can be observed when we increase the timeout for problems of size 15 in the figure 1, and that the savings illustrated in the table 1 are conservative.

Table 2. Cumulated time for 50 QCP of size 15 every 5% with a timeout of 100 s.

Algorithm	Time	Solved
$FC(vdom+)$	226	998
$IB(vdom+)$	226	998
$LDS(vdom+)$	4	1000
$PRB(vdom+)$	3	1000

Figure 1 details the performance of Forward Checking, Iterative Broadening, Limited Discrepancy Search and Progressive Random Broadening for QCP of size 15 with different time thresholds, and different value ordering heuristics.

The first two diagrams of figure 1 give the search time and the number of problem solved, every 5% for 50 QCP of order 15 at each percentage, with lexicographic value ordering and a timeout of 10 seconds. FC and IB follow the same pattern, and are completely dominated by LDS and PRB that have no phase transition.

The next two diagrams of figure 1 give similar information with random value ordering. Again, FC and IB follow similar patterns (IB is worse than FC here), and are again completely dominated by LDS and PRB that have no phase transition.

The next two diagrams of figure 1 use the *vdom+* value ordering. Here FC and IB follow exactly the same pattern and the phase transition is sharper. They are again completely dominated by LDS and PRB that have no phase transition.

The last two diagrams of figure 1 use the *vdom+* value ordering with a timeout of 100 seconds. Here FC and IB follow exactly the same pattern and the phase transition is sharper. The comparison is even better than in the previous diagrams for LDS and PRB.

Figure 2 details the performance of Forward Checking, Iterative Broadening, Limited Discrepancy Search and Progressive Random Broadening for QCP of size 30 with a timeout of 100 seconds, associated to the random and the *vdom+* value ordering heuristics. In the bottom of the figure, different values for the increment of the nb_order_max parameter have been tried, and give similar results.

The first two diagrams of figure 2 give the search time and the number of problem solved, every 5% for 50 QCP of order 30 at each percentage, with random value ordering and a timeout of 100 seconds. FC and IB follow a similar pattern. PRB and LDS are better than IB and FC for the under-constrained problems to the right of the diagrams. LDS is worse than FC and IB just after the phase transition. PRB is better than all the other algorithms everywhere (it solves more problems and uses less time).

The next two diagrams of figure 2 give similar information with the *vdom+* value ordering. Here, FC and IB have exactly the same pattern. Again LDS is worse than IB and FC just after the phase transition, and better for the under-constrained problems in the right part of the diagram. PRB is again better than all other algorithms everywhere.

The last two diagrams of figure 2 use the *vdom+* value ordering with a timeout of 100 seconds for some PRB algorithms that differs according to their increment steps for nb_max_order. They have similar patterns, even if PRB becomes slightly better when increasing the step up to 20 and becomes slightly worse for 30.

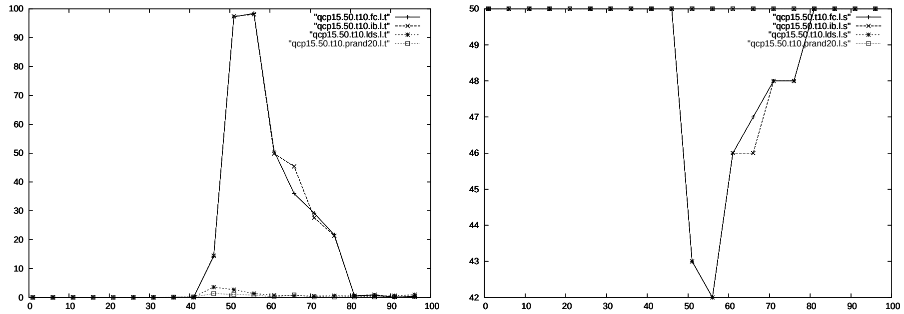
6 Conclusion and Future Work

The main result of the paper is that it is possible to improve the problem solving time of a constraint satisfaction search algorithm by progressively and randomly choosing some variables to be broadened. This algorithm performs better than forward checking, iterative broadening and limited discrepancy search for the quasi-group completion problem.

In future works, it would be interesting to test the algorithm on other domains, as well as making the probability of broadening vary according to heuristics on the size of the domains and/or on the depth of the variable.

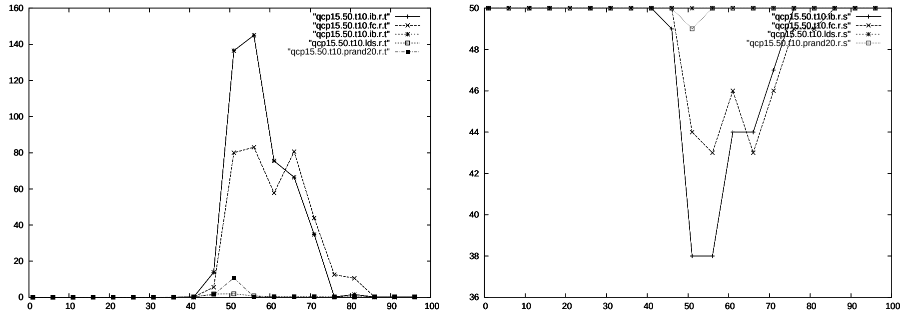
References

1. Ginsberg, M.L., Harvey, W.D.: Iterative broadening. Artificial Intelligence **55** (1992) 367–383
2. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In Mellish, C.S., ed.: IJCAI-95, Montréal, Québec, Canada, Morgan Kaufmann (1995) 607–615
3. Gomes, C.P., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. In: AAAI-98. (1998) 431–437
4. Walsh, T.: Search in a small world. In: IJCAI-99. (1999) 1172–1177
5. Achlioptas, D., Gomes, C.P., Kautz, H.A., Selman, B.: Generating satisfiable problem instances. In: AAAI/IAAI. (2000) 256–261
6. Gomes, C.P., Selman, B., Crato, N., Kautz, H.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. Journal of Automated Reasoning **24** (2000) 67–100
7. Dotu, I., del Val, A., Cebrian, M.: Redundant modeling for the quasigroup completion problem. In Rossi, F., ed.: Principles and Practice of Constraint Programming - CP 2003. Volume 2833 of Lecture Notes in Computer Science., Springer (2003) 288–302
8. Gomes, C.P., Shmoys, D.B.: The promise of lp to boost csp techniques for combinatorial problems. In: CPAIOR'02. (2002)
9. Shaw, P., Stergiou, K., Walsh, T.: Arc consistency and quasigroup completion. In: Proceedings of ECAI98 Workshop on Non-binary Constraints, Brighton. (1998)
10. Korf, R.E.: Improved limited discrepancy search. In: AAAI-96. (1996)
11. Walsh, T.: Depth-bounded discrepancy search. In: IJCAI-97. (1997) 1388–1395
12. Meseguer, P., Walsh, T.: Interleaved and discrepancy based search. In: ECAI-98. (1998) 239–243



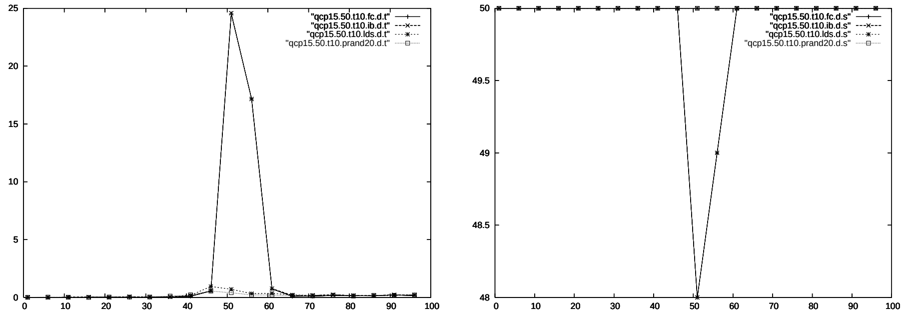
time, size 15, timeout 10s, lexicographic

\# solved



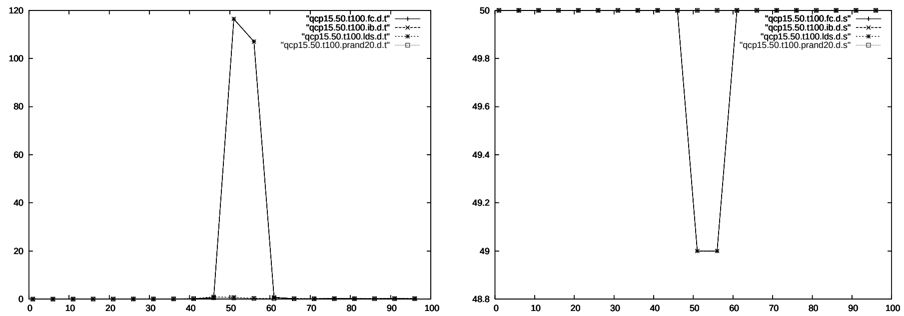
time, size 15, timeout 10s, random

\# solved



time, size 15, timeout 10s, vdom+

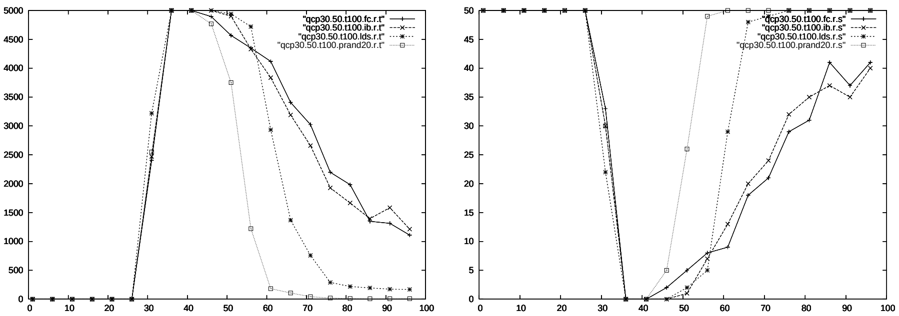
\# solved



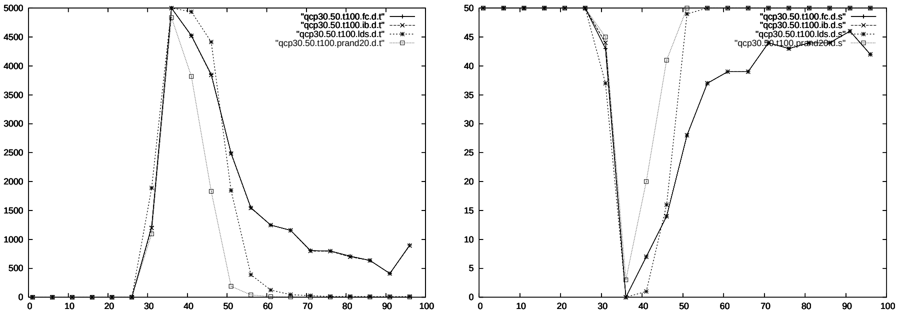
time, size 15, timeout 100s, vdom+

\# solved

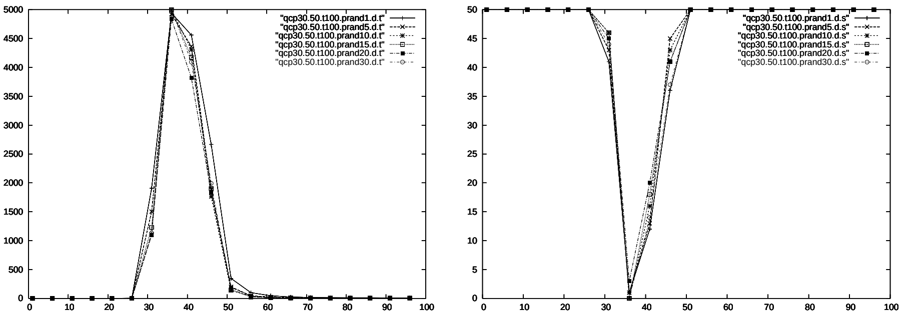
Fig. 1. Results for 50 QCP of size 15



time, size 30, timeout 100s, random   # solved



time, size 30, timeout 100s, vdom+   # solved



time, size 30, timeout 100s, vdom+, $\neq$ prb   # solved

Fig. 2. Results for 50 QCP of size 30 with a timeout of 100 s.