

On Generators of Random Quasigroup Problems

Roman Barták

Charles University, Faculty of Mathematics and Physics
Institute for Theoretical Computer Science
Malostranské nám. 2/25, Prague, Czech Republic
roman.bartak@mff.cuni.cz

Abstract. Random problems are a good source of test suites for comparing quality of constraint satisfaction techniques. Quasigroup problems are representative of structured random problems that are closer to real-life problems. In this paper, we study generators for Quasigroup Completion Problem (QCP) and Quasigroups with Holes (QWH). We propose an improvement of the generator for QCP that produces a larger number of consistent problems by using propagation through the all-different constraint. We also re-formulate the algorithm for generating QWH.

Introduction

Generators of random problems are a useful source of problem instances for testing constraint satisfaction algorithms. Writing generators for some types of problems, like Random CSP [6], is not a complicated task but it could be more complicated for other types of problems, typically for structured problems. The goal of this paper is to give all necessary information for users that would like to use quasigroup problems to test their solving algorithms. In particular, we will give a description of the algorithms generating random problem instances and we will compare the generators both in terms of quality and time efficiency.

The quasigroup problems have been first proposed as a benchmark domain for constraint satisfaction algorithms by Gomez and Selman [3]. The basic idea of the benchmark domain is to find a completion of a partial Latin square representing the multiplication table of a quasigroup. Hence, they called the problem a *Quasigroup Completion Problem* (QCP). The generator for QCP should produce a partial Latin square that can be completed to a full Latin square. However, the generator proposed in [3], which fills random values in randomly selected cells of the table, falls short on this task especially when more values should be filled in. Gomez and Selman observed a behavior of the generator similar to phase transition with satisfiable instances on one side, unsatisfiable instances on the other side, and hard instances in between. In this paper, we propose an improvement of this generator that generates a larger number of satisfiable instances so more instances are available for the tested solvers. This generator preserves the phase transition behavior but it generates satisfiable instances on both sides and it makes the phase transition crispier.

The difficulty of the above described generators is that they do not guarantee production of satisfiable instances only. This complicates usage of such generators for testing incomplete solving algorithms because when the solving algorithm did not find a solution, it is not clear whether the reason is that no solution exists or the algorithm did not find it. In the first case (no solution exists) the algorithm can be “glorified” for doing a good job, in the second case, the algorithm can be blamed for being incomplete. Therefore another benchmark domain based on quasigroups has been proposed in [1] that guarantees generation of satisfiable instances. This benchmark domain uses the same idea as QCP, that is completing a partially filled Latin square, but it differs in the way how the incomplete Latin square is obtained. The idea is to punch holes into a randomly generated complete Latin square so the obtained partial Latin square can surely be completed. Hence, this benchmark domain is called *Quasigroups With Holes* (QWH). Opposite to the generators for QCP, the generators for QWH are non-trivial and they are based on strong theoretical results presented in [5]. Unfortunately, the paper [1] proposing QWH as a benchmark does not provide all the details on generating QWH problems and the interested reader must go in [5]. In this paper, we will present the algorithm for generating instances of QWH problems as described in [1,5] and we will propose its reformulation that, in our opinion, is more natural for implementation.

The contribution of this paper is twofold. First, we will give all the details on algorithms for generating random instances of QCP and QWH problems. Second, we will present empirical comparison of the generators so the readers can select one that suits best their needs. Together, reading this paper should simplify implementation of the generators for random quasigroup problems.

The paper is organized as follows. First, we will introduce the basic terminology on quasigroups and Latin squares. Then, we will describe the Quasigroup Completion Problem, we will present the generator for QCP, and we will propose some improvements of this generator. After that, we will describe the Quasigroups With Holes (QWH) problem, we will formulate the algorithm to generate such problems according to [5], and we will propose a reformulation of the algorithm that is, in our opinion, more natural for implementation. The paper is concluded by an experimental evaluation of the generators where we will compare quality and time efficiency of the generators and will show some interesting features of the QWH generator.

Quasigroups and Latin Squares

A *quasigroup* is an ordered pair (Q, \bullet) , where Q is a set and \bullet is a binary operation on Q such that the equations $a \bullet x = b$ and $y \bullet a = b$ are uniquely solvable for every pair of elements a, b in Q . The cardinality of the set Q is called an *order* of the quasigroup. Let N be the order of the quasigroup Q then the multiplication table Q is a table of size $N \times N$ such that the cell at the coordinates (x, y) contains the result of the operation $x \bullet y$ (for simplicity reasons we expect Q to be a totally ordered discrete set and so the rows and columns of the multiplication table can be indexed by the elements of Q). The multiplication table of the quasigroup must satisfy a property that in each row of the table, each element of the set Q occurs exactly once, and similarly in each column

of the table, each element of Q occurs exactly once (see Figure 1A). Thus, the multiplication table defines a *Latin square*.

We say that a Latin square of order N is *partial* or *incomplete* if the table of size $N \times N$ is partially filled in such a way that no symbol occurs twice in a row or in a column (see Figure 1B). If the table is filled completely then we are speaking about a *complete Latin square*. Note that it is easy to generate a complete Latin square of any order. First, we take some permutation of the elements in Q . Second, we fill the first row of the table with this permutation. Third, in each subsequent row, we shift the permutation one element to the right and the superfluous element on the right is filled in the first cell of the row (see Figure 1C). However, note that generating any Latin square of a given order with a uniform probability is a non-trivial task that we will discuss later.

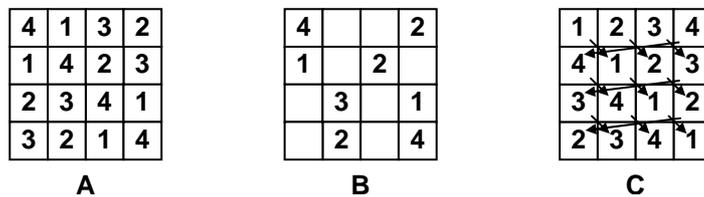


Fig. 1. A Latin square (A), a partial Latin square (B), and a process of generating a complete Latin square (C)

The problem of finding a complete Latin square can be stated as a constraint satisfaction problem in the following way. Assume, that the cells of a Latin square of order N are denoted by the variables with the domain $\{1, \dots, N\}$. Then the property of the Latin square can be described by a set of binary inequality constraints posted between every pair of variables that are either in the same row or in the same column. The constraint network for this CSP has N^2 nodes representing the variables and $N^2(N-1)$ edges representing the binary constraints (Figure 2). The network is highly structured – there are $2N$ interconnected clusters of size N (each cluster connects the variables from a single row or a single column). Moreover, there exists a path of maximal length two between any two nodes so the constraint network has a so called *small world topology*.

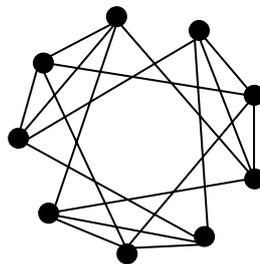


Fig. 2. A constraint network representing a Latin square of order 3

Quasigroup Completion Problem

As we showed in the previous section, a Latin square can be modeled as a CSP so it can serve as a benchmark domain for constraint satisfaction algorithms. However, we also sketched a simple algorithm to find a complete Latin square so such a benchmark is not very challenging. Gomez and Selman [3] proposed a new benchmark based on Latin squares called a *Quasigroup Completion Problem (QCP)*. In this problem, a partial Latin square is given and the task is to determine whether the empty cells can be filled in such a way that we obtain a complete Latin square. The problem is parameterized by the order of a Latin square and by the number of filled cells. Formally, the Quasigroup Completion Problem is described by a pair $\langle N, p \rangle$, where N is an order of the Latin square to be completed and p is a *filling ratio*, that is a ratio between the number of pre-filled cells and the total number of cells (N^2). Note that by filling some of the cells in the table, we introduce perturbations into the structure of constraint network which makes the problem closer to real-world instances. Thus, the Quasigroup Completion Problem bridges the gap between the purely random problems like a Random CSP [6] and the highly structured problems.

The question is how to select the cells to be filled for a given QCP $\langle N, p \rangle$. One possible model could be selecting the cell to be filled with the probability p . Let us call it a *model A* similarly to the classification used for Random CSPs [6]. Another possibility is to select exactly $\lfloor pN^2 \rfloor$ cells to be filled, where $\lfloor X \rfloor$ means the closest (to X) integer between X and 0. Let us call it a *model B*. In this paper we will study the model B, where the cells to be filled are selected randomly and uniformly. We use a random generator that selects uniformly and randomly $\lfloor pN^2 \rfloor$ different elements from the set $\{0, \dots, N^2-1\}$. Each such element z represents a position in the Latin square of order N that can be described by the coordinates $\langle 1+\lfloor z/N \rfloor, 1+(z \bmod N) \rangle$. Figure 3 shows the encoding used to identify the cell in the table.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Fig. 3. A linear encoding of the positions of cells in a Latin square or order 4

Another open question is how to select a value to be filled in a given cell. The basic requirement is that the values in cells in each row and in each column must be different. So, when selecting a value for the cell in the position $\langle x, y \rangle$, this value must be different from the values already filled in the cells of the row x and in the cells of the column y . We propose the following simple technique based on constraint propagation through binary inequalities. The Latin square is modeled as a CSP as described in the previous section, that is there are binary inequalities between the variables of the same row and of the same column. For a cell to be filled (the cell selection process is described in the previous paragraph), we select randomly a value

from the current domain of respective variable. Then the problem is made arc consistent which means that the value is removed from the variables of the same row and of the same column. Consequently, when selecting a value for a next cell, the domain contains only the values that are different from already filled values in the same row and in the same column. This technique ensures that only valid Latin squares are generated, that is no symbol occurs twice in a row or in a column. However, the constraint formulation of the generator also implies that there is no guarantee that a Latin square is found. We have a CSP and we are using randomized backtracking with no backtracks. Arc consistency is not a complete technique so if a wrong value is selected (randomly) then it is possible to obtain an inconsistent square. Because there are no backtracks, it is not possible to recover from such a situation so the generator cannot produce a valid Latin square. Figure 4 shows such a situation. One may say that this situation occurs only when a generator is based on constraint propagation but note that if any generator will try to assign a value to the variable whose domain is made empty via constraint propagation then no such value exists and so no valid partial Latin square can be generated.

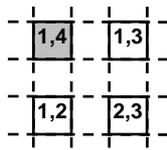


Fig. 4. The problem of simple QCP generators. If the value 1 is randomly selected for the top left cell then values 2 and 3 are removed from the bottom right cell so the problem is inconsistent and no complete Latin square exists.

When looking at Figure 4 we can see that if the value 4 is selected for the top left cell then the above problem does not occur. Therefore, we propose to enhance the generator by allowing a *shallow backtracking* that can try another (randomly selected) value after a failure. This process is repeated until a value is found or the domain is made empty. It is still possible that no value for the variable is found so this technique does not guarantee finding a valid Latin square but the hope is that it increases chances to find one. Unfortunately, as the experiments showed, this technique does not increase the number of generated valid instances (on average). Note that the generator should produce the random problems fast so its complexity should not be exponential. Therefore, we cannot use full backtracking (probably incomplete search might be used but we did not try it yet).

Another option how to improve chances of finding a value for the variable is strengthening constraint propagation that will remove more inconsistent values from the domains. There is a natural way how to strengthen propagation in the constraint model for Latin squares – using the all-different constraint by Régin [7]. So our second proposal is to use the same generator as described above but substituting the cliques of binary inequalities by the all-different constraints. As the experiments showed this improvement significantly increases the number of generated valid Latin squares especially for larger filling ratios.

Quasigroups with Holes

As we already mentioned, the main problem of QCP is that the generators cannot guarantee production of satisfiable instances. This could cause problems when evaluating incomplete solving techniques because if the technique did not find the solution then it is not clear whether this is because the problem had no solution or the technique did not find an existing solution. In the previous section we proposed a method that increases the number of satisfiable instances via strengthening constraint propagation. However, this method still does not guarantee satisfiability. It would be possible to accompany the proposed generator by a complete search that filters the unsatisfiable instances. Still, the problem is that for some parameters the generator does not produce a valid instance and hence no satisfiable instance is available for evaluation. This happens typically in the area where the hardest problems settle (see the section on experiments) so it would be beneficial if the generator produces satisfiable instances directly. Surprisingly, it is often difficult to develop a direct generator for satisfiable instances only. The problem with the generators producing only satisfiable instances is that the generator should not be biased in the sense that the generator should produce any satisfiable instance with a uniform distribution. Therefore, the simple generator of complete Latin squares described in the second section is not appropriate because it produces Latin squares with a specific structure only (and hence, completing such Latin squares is not a difficult task).

The paper [1] proposes a direct generator for satisfiable quasigroup problems. The idea is to generate a complete Latin square to which a fraction of holes is punched. The resulting incomplete Latin square is then guaranteed to be satisfiable. This problem is called *Quasigroups With Holes* (QWH).

The problem of generating uniformly distributed Latin squares is non-trivial. Actually, the generator is not described in [1] and the reader is referenced to the paper by Jacobson and Matthews [5] which describes the method and gives a theoretical justification. In the next paragraphs, we will survey the method by Jacobson and Matthews in an algorithmic form and then we will reformulate the algorithm to work directly with the Latin squares.

Original generator

Jacobson and Matthews [5] proposed a method for generating uniformly distributed random Latin squares. The idea is to see the Latin squares as nodes in a graph where the edges describe transformations between the Latin squares. They proved that the diameter of the graph is $4(N-1)^2$ so the minimal distance between two Latin squares is no greater than $4(N-1)^2$. Consequently, it is possible to obtain any Latin square from a given Latin square in $4(N-1)^2$ moves. Thus a generator can be conceived as follows. We start with a random Latin square generated for example by the method described in the second section (Figure 1C) and after performing $4(N-1)^2$ moves we should obtain any Latin square with uniform probability.

To simplify the description of moves, Jacobson and Matthews proposed to extend the graph by nodes describing so called improper Latin squares where the condition of a Latin square is “little” violated (see below). Then the diameter of the graph and

hence the minimal distance between two (proper or improper) nodes is $2(N-1)^3$ [5]. They represent the Latin square of order N by a contingency table f of size $N \times N \times N$ that contains $\{0,1\}$ values only. The condition on a Latin square (in each row and in each column, each element appears exactly once) is then equivalent to the formulas:

$$\forall x, y \in \{1, \dots, N\} \sum_{z \in \{1, \dots, N\}} f(x, y, z) = 1 \quad (a)$$

$$\forall x, z \in \{1, \dots, N\} \sum_{y \in \{1, \dots, N\}} f(x, y, z) = 1 \quad (b)$$

$$\forall y, z \in \{1, \dots, N\} \sum_{x \in \{1, \dots, N\}} f(x, y, z) = 1 \quad (c)$$

Basically, x and y describe the coordinates of the cell and z describes the element in the cell (x,y) if $f(x,y,z)=1$. So the formula (a) says that exactly one element is filled to the cell (x,y) , the formula (b) says that the element z appears exactly once in the row x , and the formula (c) says that the element z appears exactly once in the column y . We call a Latin square with the above (proper) contingency table a *proper* Latin square. An *improper* Latin square is defined by the (improper) contingency table satisfying the conditions (a)-(c) but allowing exactly one element of the contingency table to contain the value -1.

Now, it is easier to formulate the moves as operations over (proper and improper) contingency tables. Assume that we start with a proper contingency table. We select randomly a cell of f such that $f(x,y,z) = 0$ and we will try to increase this value by one which is equivalent to assigning the value z to the cell (x,y) . Each line in f containing the cell (x,y,z) must hold a cell filled by one according to (a)-(c). Let x' , y' , and z' be the indexes of these lines. These coordinates define a sub-cube in the contingency table with nodes at (x,y,z) , (x,y,z') , (x,y',z) , (x',y,z) , (x',y,z') , (x',y',z) , (x,y',z') , and (x',y',z') (see Figure 5). If we increase the value in $f(x,y,z)$ by one then we need to decrease the values in $f(x,y,z')$, $f(x,y',z)$, $f(x',y,z)$ by one to keep the conditions (a)-(c) valid. Next, the values in $f(x',y',z)$, $f(x,y',z')$, $f(x',y,z')$ must be increased by one and finally the value in $f(x',y',z')$ must be decreased by one. If all these operations are performed then visibly the conditions (a)-(c) hold again. However, it may happen that the value in $f(x',y',z')$ will become -1, in the case that $f(x',y',z')=0$, but this will be the only cell with a negative value (see Figure 5).

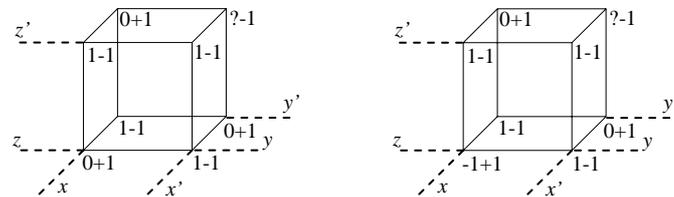


Fig. 5. A plus/minus one move in the proper (left) and improper (right) contingency table

Notice that if we start with a cell such that $f(x,y,z) = -1$ (so the original contingency table is improper) then we can perform the same set of operations as above and again

we will obtain either a proper or improper contingency table (Figure 5 right). So the above described mechanism specifies moves between proper and improper contingency table. Jacobson and Matthews [5] showed that on average after N such moves we will obtain a proper contingency table describing a Latin square of order N . Figure 6 shows the algorithm for a single move. According to [5] we propose to do at least N^3 such moves and stop when a proper contingency table is obtained.

```

move
  find x,y,z s.t.
    if f is improper then f(x,y,z)=-1
    if f is proper then f(x,y,z)=0
  find x',y',z' s.t. f(x',y,z)=f(x,y',z)=f(x,y,z')=1
  // if f is proper then these points are unique
  // if f is improper then there are two choices
  // for each point, select one point randomly
  f(x,y,z)++
  f(x,y,z')--
  f(x,y',z')++
  f(x,y',z)--
  f(x',y,z')++
  f(x',y,z)--
  f(x',y',z)++
  f(x',y',z')--
end move

```

Fig. 6. The algorithm for move between proper and improper contingency tables

Reformulated generator

In the previous section we presented the algorithm for moves between proper and improper contingency tables. Notice that if the contingency table is improper, which happens when $f(x',y',z')$ becomes -1 , then the next move starts with $f(x',y',z')$ which will be increased by one. The improper contingency table describes a situation when two values, z and the original value in (x',y') , are assigned to the cell (x',y') at the same time. According to the above observation, we know that in the next move the original value in (x',y') or z will be unassigned and so we propose to postpone assignment of z to the cell (x',y') to the next move. Before assigning the value we will check whether the value in (x',y') is z' . If this is true then we obtained a proper Latin square and we can stop the sequence of improper moves (so z is not assigned to the cell which is equivalent to assigning it and unassigning it in the next step). Otherwise we assign the value z to the cell (x',y') , we took the original value in this cell and “propagate” it further. Figure 7 describes how the values are moved between the cells.

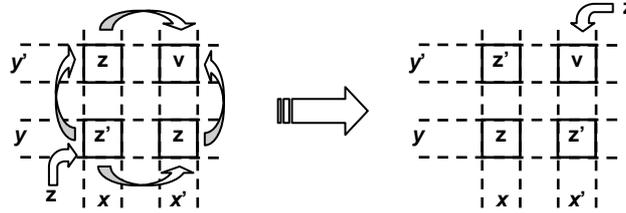


Fig. 7. Shifting values in a Latin square when the value z should be placed to position (x,y)

The above idea can now be encoded using the data structures describing directly a Latin square instead of its contingency table. Figure 8 shows the algorithm for moving between the proper Latin squares directly. The move is started with a random position (x,y) and a random value z to be placed there: `proper_move(x,y,z,z)`. When the procedure stops, a proper Latin square is obtained and another random move can be started. According to [5] we propose to call the procedure `proper_move` at least N^3 times (including the recursive calls inside `proper_move`) so every Latin square can be obtained with uniform probability.

```

proper_move(x,y,z,v)
  z' ← table(x,y)
  if z'=v then return
  y' ← a position (column) of cell with v in the row x
  x' ← a position (row) of cell with v in the column y
  // if z=v then x' and y' are unique
  // otherwise there are two such positions,
  // one position is selected randomly
  table(x,y) ← z
  table(x,y') ← z'
  table(x',y) ← z'
  proper_move(x',y',v,z')
end proper_move

```

Fig. 8. The algorithm for move between proper Latin squares

Experimental results

We have implemented the proposed generators using the `clpfd` library [2] of SICStus Prolog version 3.11.1. All presented results were accomplished under Windows XP Professional on 1.7 GHz Mobile Pentium-M 4 with 768 MB RAM. The running time is measured in milliseconds via the `statistics` predicate with the `walltime` parameter [8]. For each problem we generated 100 random instances, the average results are presented.

Generator Quality

Generators of random problems are often expected to produce satisfiable instances because such instances can be solved by the tested algorithms (solvers). In particular, if the generator is not able to produce a problem instance of given parameters then the solver cannot be tested for given problem parameters at all. Thus, we propose to measure the quality of the generator by the number of satisfiable instances that is by the number of generated instances that have a solution. In our experiments, we approximated the generator quality by the number of consistent instances which is faster to count than counting the satisfiable instances. Note that if a problem is not (locally) consistent then it is not satisfiable. Moreover, the stronger consistency technique we use, the closer approximation we get. For testing consistency, we used (generalized) arc consistency with the all-different constraints posted for the variables in each row and in each column. We used the all-different constraint by Régin [7] because it achieves better domain pruning than a set of binary inequalities so it can discover more inconsistencies and so it seems to be a good compromise between the consistency level and the time complexity. Note also, that sometimes the generator is not able to produce a valid instance at all. We say that the instance is valid if it contains a given number of filled cells ($\lfloor pN^2 \rfloor$, where p is the filling ratio) and no symbol occurs twice in a row or in a column. If the instance is not valid then we assume it to be inconsistent. In our first experiment, we measured the number of consistent instances introduced by the generator for different filling ratios.

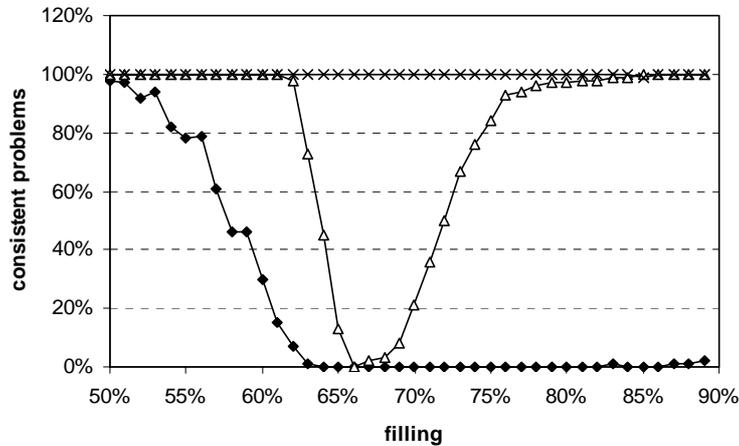


Fig. 9. The relative number of generated consistent problems as a function of the filling ratio (in percent) for the quasigroup problems of order 30 (♦: QCP-orig, Δ: QCP-alldiff, ×: QWH)

Figure 9 shows the results for the quasigroup problems of order 30 and different filling ratios. We focus on the area with hard problems – a so called phase transition. We can see that the quality of the original generator of the Quasigroup Completion Problem degrades earlier and since the filling ratio 0.64 the generator is not able to produce a consistent instance at all. This corresponds to observations in [3] where the

authors measured the number of unsolvable instances (note also that they use a complement to our filling ration that is a number of holes in the table). However, this does not mean that solving QCP with a large number of filled cells is a hard problem. Actually, as our experiments showed the problem is that the original generator is not able to generate satisfiable instances in the area with a large filling ratio. When we use a QCP generator with the all-different constraint [7] then we can generate consistent instances even in this area. The reason is that the all-different constraint propagates more than a set of binary inequalities so it can prevent selection of a “bad” value during filling the cells. Notice also, that the QCP generator with the all-different constraint produces a larger number of consistent instances even for smaller filling ratios. This feature makes the phase transition area crispier so we can generate harder problem instances for testing the solving algorithms.

We also accompanied both QCP generators with the shallow backtracking feature as described in the previous sections. However, we obtained the number of consistent instances identical to the number of consistent instances produced by the generators without shallow backtracking (the versions with shallow backtracking are not included in Figure 9). Thus, shallow backtracking does not seem to contribute to the generator quality.

Finally, the QWH generator produces consistent instances only. This is not surprising because the QWH generator produces satisfiable instances.

Time efficiency

The generators of random problems should run fast because the core of the user’s experiment is usually testing the constraint satisfaction algorithms so the users do not want to waste time by generating the problems. Note that random problems are not usually generated off-line so their runtime might be an important factor. In our second experiment, we compared the runtime of the proposed generators. In particular, we measured the runtime as a function of the filling ratio (Figure 10) and as a function of the order of a Latin square (Figure 11). Time spent in the generator for producing both valid and invalid instances is measured.

Figure 10 shows the results for the quasigroup problems of order 30 and different filling ratios. Again, we focused on the area with hard problems. The QCP generators are filling the required number of cells so the runtime is increasing with the larger filling ratio. The runtime of the QWH generator is (almost) independent of the filling ratio because the generator spends most of the time exploring complete Latin squares (which is independent of the filling ratio) and the time to punch the holes in the table is neglected there. The bad news are that the runtimes for QWH and QCP-alldiff generators are significantly (7.5-10×) larger than the runtime for the original QCP generator. Moreover, we can see that the QWH generator has better runtime than the QCP-alldiff generator and the difference increases with the larger filling ratio. Because the QWH generator produces satisfiable instances, it will be probably preferred over the QCP-alldiff generator there.

The runtime for QCP generators with shallow backtracking is identical to the runtime of generators without shallow backtracking (the versions with shallow backtracking are not included in Figure 10).

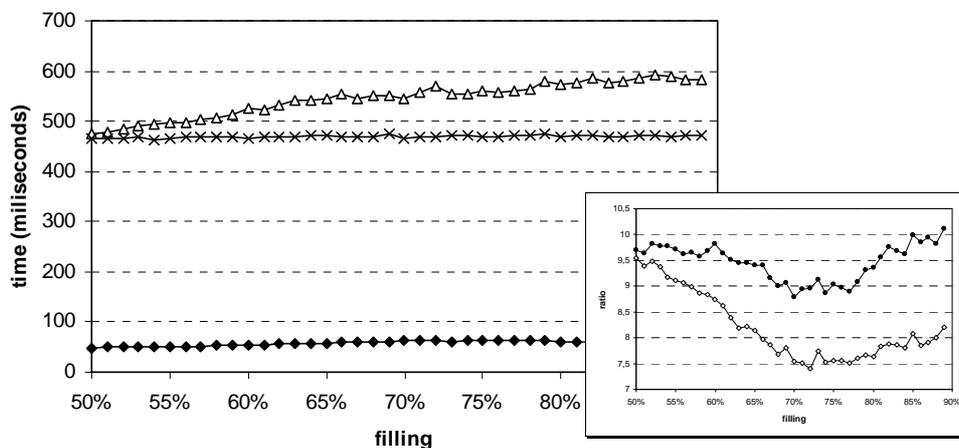


Fig. 10. The time (in milliseconds) to generate a quasigroup problem of order 30 as a function of the filling ratio (in percent) (\blacklozenge : QCP-orig, \triangle : QCP-alldiff, \times : QWH). A small graph on the right shows a ratio between the running times as a function of the filling ratio (\bullet : QCP-alldiff/QCP-orig, \diamond : QWH/QCP-orig).

Figure 11 compares the runtimes of the generators on problems with a fixed filling ratio 0.6 and with changing order of a Latin square. We have selected the filling ratio 0.6 because it is close to the phase transition, however, we performed experiments with other filling ratios (namely 0.5, 0.7, and 0.8) and the results were very similar.

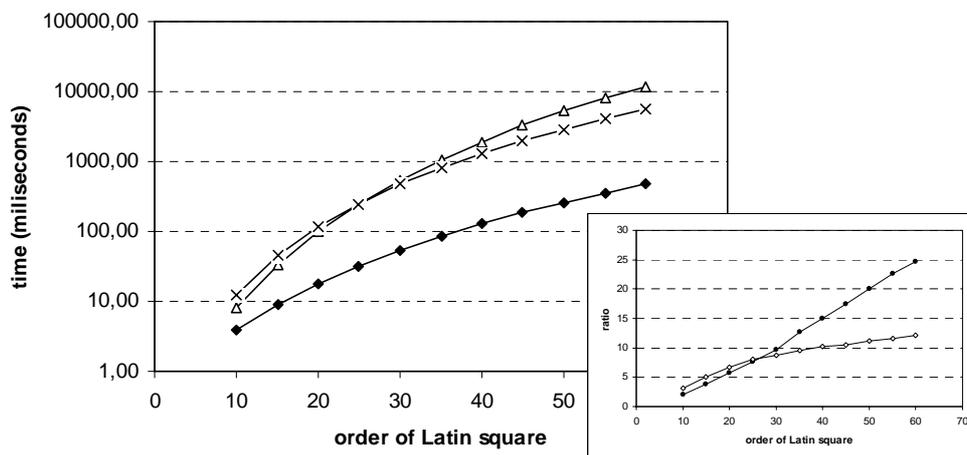


Fig. 11. The time (in milliseconds, a logarithmic scale) to generate a quasigroup problem with the filling ratio 0.6 as a function of the order (\blacklozenge : QCP-orig, \triangle : QCP-alldiff, \times : QWH). A small graph on the right shows a ratio between the running times as a function of the order (\bullet : QCP-alldiff/QCP-orig, \diamond : QWH/QCP-orig).

For all the generators the runtime is increasing significantly with the increased order of a Latin square but the increase is less than exponential. We know that the theoretical time complexity for the QWH generator is cubical which is confirmed by our experiments. From Figure 11, we can deduce that the time complexity of the QCP-alldiff comparator is close to cubical and that since the order 25, it is more efficient to use the QWH generator that is slightly faster and, moreover, it produces satisfiable instances. Position of this border point, a point when QWH becomes more efficient than QCP-alldiff, depends on the filling ratio. The border point is higher (30 for 0.5, see Figure 10) for smaller ratios. Definitely, for larger problems (the order 30 and more), the QWH generator is more efficient than the QCP-alldiff generator in the area of phase transition. For smaller problems, the QCP-alldiff might still be useful. From Figure 11, we can also see that the QCP-alldiff generator is linearly slower than the QCP-orig generator.

QWH features

Even if the resulting problems are identical for QCP and QWH – filling the cells in the incomplete Latin square – the generators are based on very different principles. The goal of our last experiment is to study behavior of the QWH generator. Recall that the generator starts with a randomly generated Latin square and it performs steps to modify this initial Latin square. For problems of order N , at least N^3 steps are performed. In each step, some cells are exchanged in the table so the table may become improper with respect to the features of the Latin square. In particular, it is possible that some row and column may contain a pair of identical cells. The algorithm guarantees that this improper table will become proper after few steps so we measured the average number of steps between two proper tables. We also measured the average number of generated proper Latin squares during a single run of the generator.

Table 1. Some numerical features of the QWH generator

order	10	20	30	40	50	60
runtime (milliseconds)	13	118	476	1326	2946	5739
steps between proper squares	9.89	19.90	29.82	39.84	49.92	59.76
explored proper squares	102	403	906	1607	2505	3615

Table 1 shows the results of the QWH generator depending on the order of a Latin square. We have already mentioned the cubic time complexity of the generator. The interesting result is that for the problems of order N , the average number of steps between two proper squares is slightly less than N . This confirms the theoretical results from [5] where the authors proved that the upper bound on the length of the shortest sequence of steps between any two (proper or improper) squares is $2(N-1)^3$ and the upper bound on the length of the shortest sequence of proper moves between any two Latin squares is $4(N-1)^2$. From Table 1, we can also see that slightly more than N^2 proper Latin squares are explored during each run of the generator so according to [5] there is a chance to obtain any Latin square from a given randomly generated Latin square.

Conclusions

In this paper, we studied the generators for quasigroup problems. We have proposed an improvement of the generator of the Quasigroup Completion Problem based on the idea of using all-different constraints to propagate partial assignments. Experimental results confirmed that this improved generator produces more consistent instances especially in the area of phase transition. Consequently, this generator is more appropriate for preparing problem instances to be used for testing solving algorithms. Unfortunately, the runtime of this improved generator is much larger than the runtime of the original generator. The open question (for experiments) is how constraint propagation used by both generators influences the generated instances. In particular, how many variables are instantiated via constraint propagation so what is the actual filling ratio. The challenge is to find out a generator that produces a higher number of satisfiable instances without a big efficiency penalty. One of the possible ways could be using incomplete search techniques with a guaranteed runtime.

We have also reformulated the algorithm for generating Quasigroups With Holes in a such a way that it is easier to go from a proper instance to another proper instance. We believe that this formulation is more natural because it works with 2D tables describing the Latin squares rather than using a 3D table with $\{0,1\}$ values like in [5]. Anyway, as far as we know this is the first paper in the CSP literature giving the exact description of the generator for QWH. The original paper [1] on QWH did not describe the algorithm completely and the paper [5] is theoretical and the algorithm is not given there as well (even if all the details specifying the algorithm are described there).

We also performed experiments comparing the generators of QWH and QCP. The results showed that the time efficiency of the QWH generator is comparable to the time efficiency of the QCP generator with the all-different constraints. We also experimentally showed that the number of steps of the QWH generator between two proper instances corresponds to the order of the Latin square. An open question is how hard instances the generators produce. This can be measured, for example, by the number of solutions to generated problems.

Acknowledgements

This work is supported by the Czech Science Foundation under the contract no. 201/04/1102 and by the project LN00A056 of the Ministry of Education of the Czech Republic. The author has been awarded an ECCAI Travel Grant to attend ECAI'04.

References

1. Dimitris Achlioptas, Carla Gomes, Henry Kautz, and Bart Selman. *Generating Satisfiable Problem Instances*. In Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00). AAAI Press, pp. 256-261, 2000.
2. M. Carlsson, G. Ottosson, and B. Carlson. *An open-ended finite domain constraint solver*. In Programming Languages: Implementations, Logics, and Programming. Springer-Verlag LNCS 1292, 1997.
3. Carla Gomez and Bart Selman. *Problem Structure in the Presence of Perturbations*. In Proceedings of Fourteenth National Conference on Artificial Intelligence (AAAI-97). AAAI Press, pp. 221-226, 1997.
4. Carla Gomez and David Shmoys. *Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem*. In Proceedings Computational Symposium on Graph Coloring and Generalizations, 2002.
5. Mark T. Jacobson and Peter Matthews. *Generating uniformly distributed random latin squares*. Journal of Combinatorial Designs 4(6), pp. 405-437, 1996.
6. Ewan MacIntyre, Patrick Prosser, Barbara Smith, and Toby Walsh. *Random Constraint Satisfaction: theory meets practice*. In Michael Maher and Jean-Francois Puget (eds.): Principles and Practice of Constraint Programming - CP98. Springer-Verlag LNCS 1520, pp. 325-339, 1998.
7. Jean-Charles Régin. *A filtering algorithm for constraints of difference in CSPs*. Proceedings of Twelfth National Conference on Artificial Intelligence (AAAI-94). AAAI Press, pp. 362-367, 1994.
8. SICStus Prolog User's Manual, Release 3.11.1, SICS, 2004.