On Generators of Random Quasigroup Problems

Roman Barták[*]

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

Abstract. Problems that can be sampled randomly are a good source of test suites for comparing quality of constraint satisfaction techniques. Quasigroup problems are representatives of structured random problems that are closer to real-life problems and hence more suitable for benchmarking. In this paper, we describe in detail generators for Quasigroup Completion Problem (QCP) and Quasigroups with Holes (QWH). In particular, we study an improvement of the generator for QCP that produces a larger number of satisfiable problems by using propagation through the all-different constraint. We also re-formulate the algorithm for generating QWH that is much faster than the original generator. Finally, we provide an experimental comparison of all presented generators.

1 Introduction

Generators of random samples for problems are a useful source of problem instances for testing constraint satisfaction algorithms. Writing generators for some types of problems, like a Random CSP [11], is not a complicated task but it could be more complicated for other types of problems, typically for structured problems like quasigroup problems. This paper gives all necessary information for researchers that would like to use quasigroup problems as benchmarks.

The quasigroup problems have been first proposed as a benchmark domain for constraint satisfaction algorithms in [6]. The basic idea is to find a completion of a partial Latin square representing the multiplication table of a quasigroup. Hence, the problem is called a *Quasigroup Completion Problem* (QCP). The generator for a QCP should produce a partial Latin square that can be completed to a full Latin square. However, the generator proposed in [6], 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 [6] 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. Shaw et al. [14] proposed an improvement of this generator based on propagation through the all-different constraint [13]. Their algorithm generates a larger number of satisfiable instances that can be used for testing solvers. It preserves the phase transition behavior but it generates satisfiable instances on both sides and it makes the phase transition crispier.

[*] Supported by projects 1M0021620808 and MSM0021620838 of the MŠMT.

The difficulty of QCP 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 no solution exists or the algorithm is not able to find it. 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 a QCP, that is completing a partially filled Latin square, but it differs in 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). Unfortunately, the authors of QWH did not provide all the details on generating QWH problems. It is a pity because generating randomly distributed QWH problems is a non-trivial task based on strong theoretical results presented in [9].

The contribution of this paper is threefold. First, we will give all the details on algorithms for generating random instances of QCP and QWH problems so interested readers will be able to write their own generators based on the presented algorithms. Second, we will propose a reformulated algorithm for generating QWH problems that is significantly faster then the original algorithm from [9]. Last but not least, we will present an empirical comparison of all presented generators so readers can select one that best suits their needs.

The paper is structured as follows. In the next section, we will introduce the terminology on quasigroups and Latin squares. In Section 3, we will describe the Quasigroup Completion Problem and its relevance to real-world problems and we will discuss two generators of QCP. In Section 4, the ideas behind Quasigroups With Holes will be explained, the original QWH generator will be presented in detail, and the reformulated generator will be introduced. The paper is concluded by an experimental evaluation of the quality and time efficiency of the generators.

2 Quasigroups and Latin Squares

A *quasigroup* is an ordered pair (Q, •), where Q is a set and • 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 of Q is a table of size N×N such that the cell at the coordinates (x,y) contains the result of the operation $x \bullet y$ (for simplicity 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×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. We take some permutation of the elements in Q. We put it in the first row of the table, and, 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, this method does not produce every Latin square. In fact, generating any Latin square of a given order with a uniform probability is a non-trivial task [9].

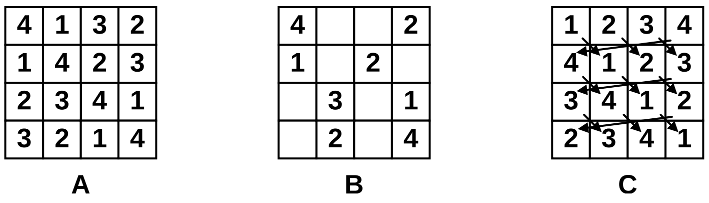

Fig. 1. A Latin square (A), a partial Latin square (B), and a simple process of generating a complete Latin square (C)

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,…,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. 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*. Nowadays the binary inequality constraints in each row and column can be encapsulated into an all-different constraint which achieves stronger pruning and makes the problem easier to solve (but still cannot solve the problem of any order [5,8]).

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. We also sketched a simple algorithm to find a complete Latin square so such a benchmark is not very challenging. Assume now, that some cells in the Latin square are pre-filled, we have a partial Latin square, and the task is to determine whether the empty cells can be filled in such a way that we obtain a complete Latin square. Gomez and Selman [6] proposed this new benchmark based on completing partial Latin squares and they called it a *Quasigroup Completion Problem* (QCP). 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 the ratio between the number of pre-filled cells and the total number of cells (N^2).

Pre-assigning some values to variables modeling the Latin square introduces perturbations in the structure of the constraint network which makes the structure similar to that found in real-world domains like scheduling and experimental design [8]. A particular real-life problem that maps directly to the above mentioned problem of completing a partially filled Latin square is the problem of assigning wavelengths to routes in fiber-optic networks [10]. Note also that the Quasigroup Completion Problem is known to be an NP-complete problem [4]. Not surprisingly, the straightforward constraint model with the all-different constraints cannot be used alone to solve instances of higher order (34 and more) [8] and more sophisticated techniques like hybrid algorithms [8] or dual models with special value selection heuristics [5] are necessary. This makes the problem non-trivial and hence interesting as a benchmark for comparing constraint satisfaction techniques. This benchmark bridges the gap between purely random problems like a Random CSP and highly structured problems.

The question now is how to generate random instances of QCP, in particular how to select the cells to be pre-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 [11]. 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 $\{0,\ldots,N^2-1\}$. Each such element z represents a position in the Latin square of order N that can be described by the coordinates $\langle 1+\lfloor z/N \rfloor, 1+(z \textit{ mod } N)\rangle$ (Figure 2).

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

Fig. 2. A linear encoding of the positions of cells in a Latin square of order 4

The second open question is how to select a value to be assigned to 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 assigned to the cells of the row x and to the cells of the column y. We propose the following simple technique based on constraint propagation through binary inequalities. Latin square is modeled as a CSP as described in the previous section using binary inequalities between the variables of the same row and of the same column. For a cell to be assigned (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 the next cell, the domain contains only the values that are different from already assigned values in the same row and in the same column. This technique mimics the behavior of the original generator from [6]. It ensures that only valid Latin squares are generated, that is no symbol occurs twice in a row or in a column. However, because of incompleteness of constraint propagation we cannot guarantee that a “completable” Latin square is found. Figure 3 shows a situation where a bad initial value selection makes filling of another cell impossible. Note that any generator attempting to generate values one by one suffers from this problem.

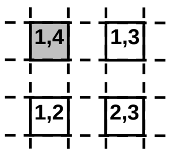

Fig. 3. The problem of simple QCP generators. If value 1 is selected for the top left cell then no complete Latin square exists.

When looking at Figure 3 we can see that if value 4 is selected for the top left cell then the above problem does not occur. Therefore, it might be useful to enhance the generator by allowing a *shallow backtracking* that can try another (randomly selected) value after an immediate failure. This process is repeated until a value is found or all values were tried. 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, our preliminary experiments showed that 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 to remove more inconsistent values from the domains. As we already mentioned, there is a natural way how to strengthen propagation in the constraint model for Latin squares – using the all-different constraint by Régin. This approach has already been proposed in [14] – we will present a detail experimental comparison of both generators later in the paper. It will show that the generator based on all-different constraints produces a higher number of satisfiable instances.

4 Quasigroups With Holes

As we already mentioned, the main problem of a QCP is that the generators cannot guarantee production of satisfiable benchmark instances which could cause problems when evaluating incomplete solving techniques. In the previous section we described a method that increases the number of satisfiable instances via strengthening constraint propagation, but this method still does not guarantee satisfiability (see the next section for experimental justification of these claims). It would be possible to accompany the proposed generator by an algorithm 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 of satisfiable instances. The problem with such generators is that they 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 Section 2 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). However, the problem of generating uniformly distributed Latin squares is non-trivial. Actually, the generator is not described in [1] and the reader is referred to the paper [9] which describes the method and gives a theoretical justification. In the next paragraphs, we will survey the method from [9], we will present the QWH generator based directly on this method, and then we will reformulate the generator to work directly with the Latin squares.

4.1 Original Generator

Jacobson and Matthews [9] proposed a method for generating uniformly distributed random Latin squares by randomly traversing a graph, where nodes correspond to Latin squares and edges describe transformations between the Latin squares. They proved that the diameter of the graph is $4(N-1)^2$, where N is the order of the Latin square. It means that the minimal distance between two Latin squares is no greater than $4(N-1)^2$ so it is possible to obtain any Latin square from a given Latin square in $4(N-1)^2$ moves. The QWH generator can be conceived as follows. We start with a Latin square generated by the method described in Figure 1C. After performing $4(N-1)^2$ random moves we obtain any Latin square with uniform probability [9]. The open question is how to perform a move, that is, how to transform one Latin square into another Latin square. We will answer this question in the following paragraphs.

To simplify 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 violated a “little” (see below). Then the diameter of the new graph and hence the minimal distance between any two (proper or improper) nodes is bounded by $2(N-1)^3$ (for a formal proof see [9]). They represent the Latin square of order N by a contingency table f of size N×N×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:

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

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

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

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. Formula (*a*) says that exactly one element is assigned to the cell (*x*,*y*), formula (*b*) says that the element *z* appears exactly once in the row *x*, and 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 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 4). 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 4).

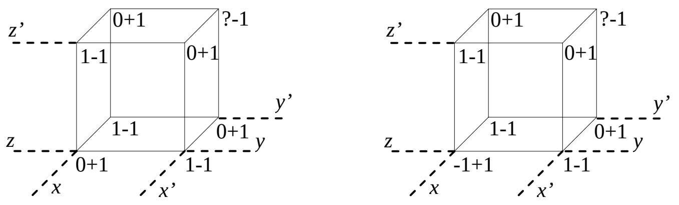

Fig. 4. A plus/minus one move in the proper (left) and improper (right) contingency table.

Notice that if we start with a cell such that *f*(*x*,*y*,*z*) = -1 (the 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 4 right). Hence the above described mechanism specifies moves between proper and improper contingency tables. Notice that the cell (*x*,*y*,*z*) is chosen randomly for a proper contingency table, while this cell is unique in the improper contingency table. Conversely, points *x'*, *y'*, and *z'* are unique in the proper table, while these points are chosen randomly in the improper table. This randomness is crucial to obtain random moves. Jacobson and Matthews showed that on average after N such random moves we will obtain a proper contingency table describing a Latin square of order N. Figure 5 shows the algorithm for a single move. By using information about the diameter of graph with nodes marked by Latin squares (see above) we propose to do at least $2(N-1)^3$ such moves and then 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
   increase f(x,y,z),f(x,y′,z′),f(x′,y,z′),f(x′,y′,z)
   decrease f(x,y,z′),f(x,y′,z),f(x′,y,z),f(x′,y′,z′)
end move
```

Fig. 5. The algorithm for move between contingency tables.

4.2 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'*) that will be increased by one. Moreover, one of the cells *f*(*x'*,*y'*,*v*) or *f*(*x'*,*y'*,*z*) will be decreased by one, where *v* is the original value at position (*x'*,*y'*). This is because the improper contingency table describes the situation when two values, *z* and the original value *v* in (*x'*,*y'*), are assigned to the cell (*x'*,*y'*) at the same time (recall, that *f*(*x'*,*y'*,*z*) has been increased by one in the step preceding this situation). To prevent appearance of two elements in a single cell we propose to postpone assignment of *z* to the cell (*x'*,*y'*) to the next move. Before assigning the value we check whether the value in (*x'*,*y'*) is *z'*. If this is true then we put *z* there so we get a proper Latin square and we can stop the sequence of improper moves. Otherwise, we also assign the value *z* to the cell (*x'*,*y'*), but we take the original value in this cell and "propagate" it further. Figure 6 describes how the values are moved between the cells.

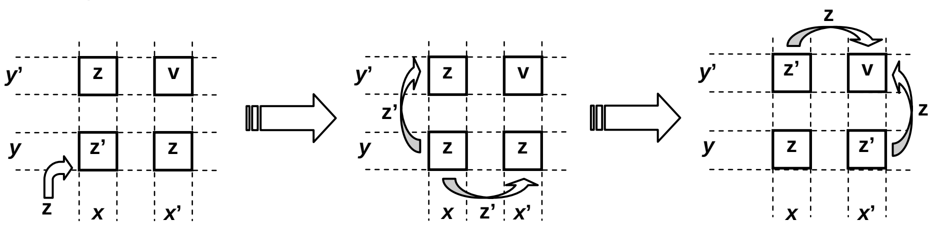

Fig. 6. Shifting values in a Latin square when the value *z* should be placed in position (*x*,*y*).

The above idea can be encoded using data structures describing directly a Latin square instead of its contingency table. Figure 7 shows the algorithm for moving between 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. Notice that if the Latin square is improper then there are two positions in the row *x* and two positions in the column *y* where the value *v* is located. In such a case, one position in the column and one position in the row are selected randomly. If the position is selected deterministically, for example the first found position, then the algorithm starts cycling! As in the original generator we propose to call the procedure `proper_move` at least $2(N-1)^3$ 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 table(x,y) ← z, 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. 7. The algorithm for move between proper Latin squares.

5 Experimental Results

We have implemented the presented generators using the clpfd library [3] of SICStus Prolog version 3.11.2. All presented results were accomplished under Windows XP Professional on 1.8 GHz Pentium 4 with 512 MB RAM. The running time was measured in milliseconds via the `statistics` predicate with the `walltime` parameter. The results of a hundred runs are presented. QCP-orig is the original QCP generator [6], QCP-alldiff is the generator using all-different constraints [14], QWH-orig is the QWH generator using contingency tables (Section 4.1), and QWH-new is the reformulated QWH generator (Section 4.2).

5.1 Generator Quality

Generators of random problems are expected to produce problems in the whole spectrum of their parameters. In our first experiment, we measured the number of generated partial Latin squares relative to the number of attempts to generate a problem. Recall, that the generator should produce a partial Latin square, namely no symbol occurs twice in a row or in a column, with a given filling. Figure 8 shows the result for Latin squares of order 30 and different filling ratios. Notice that the original QCP generator falls short on the task of generating problems where more pre-filled cells are requested. Actually, the generator is not able to produce any instance when the filling ratio is greater than 72%. This is not surprising because the more cells should be pre-filled the higher probability is that no value can be found for some cell (see Figure 3). A similar behavior can be observed for the QCP-alldiff generator but thanks to stronger propagation via all-different constraints the chances to select a consistent value increases and hence the generator is still able to produce some instances. It is a pity that the papers [6,14] proposing these generators did not mention this feature, probably because the authors used Latin squares of small orders (below 20) where this behavior cannot be observed. For the sake of completeness, let us highlight that the QWH generators always produce a problem instance.

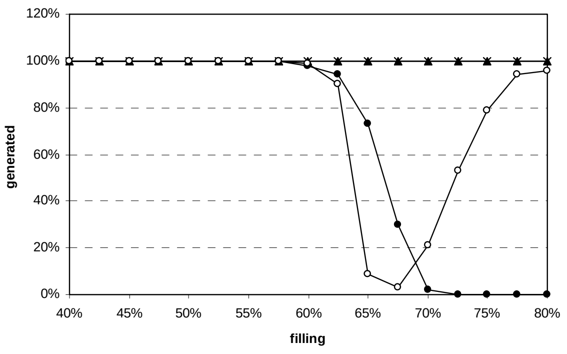

Fig. 8. The relative number of generated problems for the quasigroup problems of order 30 (●: QCP-orig, ○: QCP-alldiff, ▲: QWH-orig, ✕: QWH-new).

The second feature that we focused on is the "hardness" of the generated problems. We used the presented generators to produce partial Latin squares of order 30 which is the border where the quasigroup problems are non-trivial but still solvable by standard constraint satisfaction techniques [8]. To solve the problem we used a standard MAC algorithm with the constraint model using all-different constraints, "smallest domain first" variable selection, and "minimal value first" value selection. We used a time limit of 2 minutes to solve each problem (there are some very hard instances that would prevent finishing experiments in a reasonable time if timeout is not used).

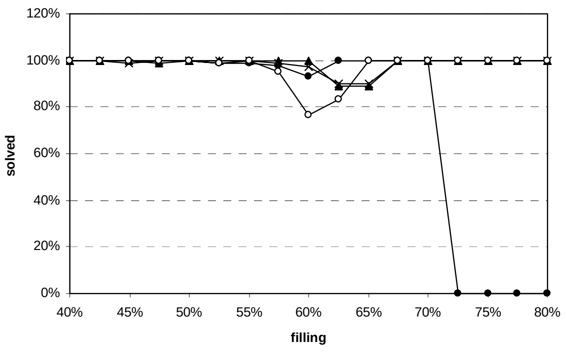

Fig. 9. The relative number of solved problems for the quasigroup problems of order 30 (●: QCP-orig, ○: QCP-alldiff, ▲: QWH-orig, ✕: QWH-new).

Figure 9 shows a relative number of solved instances as a function of filling ratio. As we can see most of the generated problems are solvable within the 2 minutes timeout but there are some instances around 62% that were not solved. This is a first indication where the hard problems might settle but it does not show yet how hard the problems are. No problem is generated by QCP-orig for filling rations above 72% and hence no problem is solved there.

We compare hardness of the generated problems by measuring runtime of the above described straightforward solver when solving the problems generated by the studied generators. Figure 10 shows median runtime to solve the generated problems. By solving the problem we understand finding a completion of the partial Latin square or proving that no completion exists. This experiment brought some surprising results. First, the phase transition area is shifted for the QWH generators towards the area with a higher filling ratio (in comparison with the QCP generators). Second, the QCP-alldiff generator produces the hardest to solve instances. This could be caused by using the all-different constraints both inside the QCP-alldiff generator and inside the solver, but we have no evidence of this (probably trying another solving approach might show whether the generated instances are hard in general). Finally, notice that the QCP-orig generator produced quite easy problems. We have observed the above mentioned features for Latin squares of other orders too.

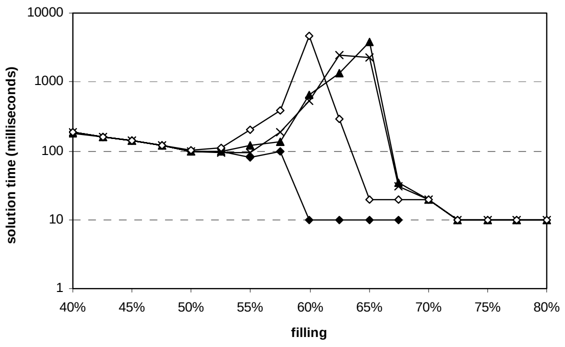

Fig. 10. Median solution time in milliseconds (logarithmic scale) for the quasigroup problems of order 30 (●: QCP-orig, ○: QCP-alldiff, ▲: QWH-orig, ×: QWH-new).

As we already mentioned, it should be clear whether the generator produces satisfiable instances or not. Production of satisfiable instances is especially important when the problems are used to compare incomplete algorithms like local search techniques or incomplete depth-first search techniques [2]. In the next experiment we measured the number of satisfiable instances among the solved problems. Figure 11 shows the relative number of satisfiable instances for Latin squares of order 30 and different filling ratios. QWH generators are guaranteed to produce satisfiable instances; the experiment just confirmed this feature. Hence these generators are appropriate for providing instances to compare incomplete algorithms. The behavior of QCP-orig generator with satisfiable instances on one side and unsatisfiable instances on the other side has already been presented at [6]. However, taking in account Figure 9, we can deduce that no satisfiable instance is generated for larger filling ratios simply because no instance is generated there. Hence the conclusions in [6] are a bit misleading because the readers might expect that QCP-orig produces unsatisfiable instances for larger filling ratios which is not true in general (especially for higher order of Latin squares). The number of satisfiable instances produced by QCP-alldiff is also decreasing around the phase transition area but it increases again for large filling ratios. Our other experiments (not presented here) showed that the area with a smaller number of satisfiable instances enlarges with increasing order of the Latin square. Nevertheless, QCP-alldiff might still be appropriate for generating problems used to compare complete algorithms.

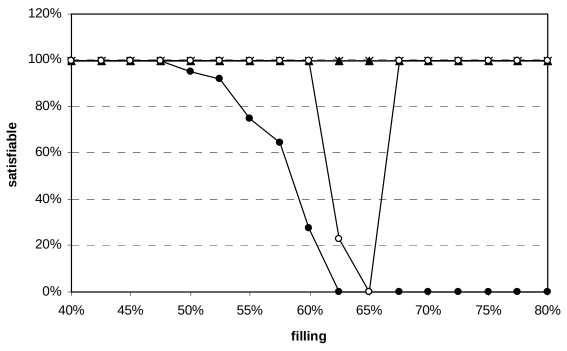

Fig. 11. The relative number of satisfiable instance for the quasigroup problems of order 30 (●: QCP-orig, ○: QCP-alldiff, ▲: QWH-orig, ×: QWH-new).

In our last experiment, we tried to estimate how complicated the generated problems are if the constraint model with all-different constraints is used. In particular, we measured the number of cells that have a value after the all-different constraints are posted and propagated but before search is started. Mean values among the consistent problems are presented in Figure 12. The dashed line indicates the initial filling produced by generators so the curves above this line indicate that additional values are deduced by the initial constraint propagation using the all-different constraints. Notice that for the filling ratio smaller than 60%, no values for additional cells were deduced while for the filling ratio greater than 70%, the values of all the variables were set using constraint propagation (so no search is necessary to solve the problem). We can see that the initial constraint propagation deduced more values for the problems produced by the QCP-all generator in comparison to the QWH generators. This is probably caused by using the all-different constraints during generation. Hence the QCP-alldiff produces instances with a larger number of pre-filled cells than requested. As proposed in [14] the intended filling can be achieved by measuring the total number of instantiated variables (including those instantiated through propagation) and stopping the generation process when this number is equal to the required filling. However, this technique makes the generator “less random” because some cells are filled by propagation rather than randomly. Moreover, this technique is not applicable to QWH generators that do not use propagation. If we use an assumption that stronger initial pruning means that the problems are easier for solving using the CP technology then the QWH generator produces harder problems in the phase transition area. This fits our observation from Figure 10, but recall that the phase transition area is shifted to smaller filling ratios for QCP-alldiff.

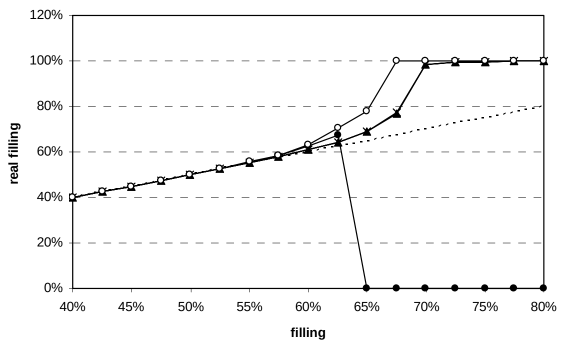

Fig. 12. The relative number of pre-filled cells using all-different constraints for the quasigroup problems of order 30 (●: QCP-orig, ○: QCP-alldiff, ▲: QWH-orig, ×: QWH-new).

5.2 Generator Efficiency

Sometimes the generators of benchmark problems run off-line so they produce problems that are put into benchmark libraries. Nevertheless, in case of random samples of problems, the generators are frequently used on-line to generate problems that are used immediately to test the solvers. In this second case, it is desirable for the generator to be fast (the users do not want to waste time by generating the problems).

We measured the runtime of studied generators to show how appropriate the generators are for on-line experiments. Figure 13 shows the runtime as a function of the filling ratio and Figure 14 shows the runtime as a function of the order of a Latin square. Visibly the better quality of the QCP-alldiff generator is paid-off by longer runtime. Moreover, the runtime of the QCP-alldiff generator increases faster than the runtime for the new QWH generator and from the order 50, it is actually slower. Consequently, the QCP-alldiff generator pays-off only for smaller order of the Latin square which also takes in account our discussion from the previous section. Recall that QCP-alldiff seems to produce the hardest to solve instances (Figure 10) so we believe that this generator is still appropriate for comparing complete solving techniques. Despite the fact that the original QCP generator is very fast, we do not recommend its usage simply because it produces less problem instances and the generated problem instances are order of magnitude easier to solve (Figure 10) in comparison to other presented generators. The runtime of QWH generators is slower than the original QCP generator, but recall that all instances produced by the QWH generators are satisfiable, which makes the QWH generators the only choice for testing incomplete solving techniques. Notice also that the reformulated QWH generator is about two times faster than the original QWH generator.

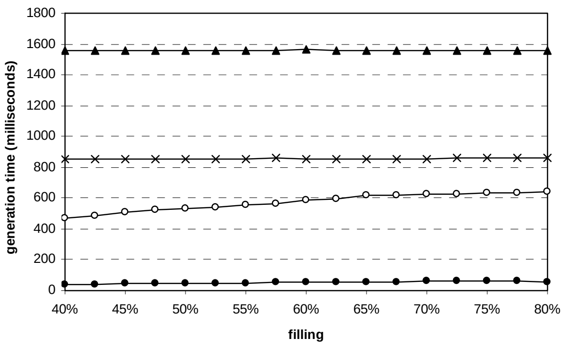

Fig. 13. The time (in milliseconds) to generate a quasigroup problem of order 30 and variable filling (●: QCP-orig, ○: QCP-alldiff, ▲: QWH-orig, ✕: QWH-new).

Figure 14 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 filing ratio 0.6 because it is within the phase transition region, however, we performed experiments with other filling ratios and the results were similar.

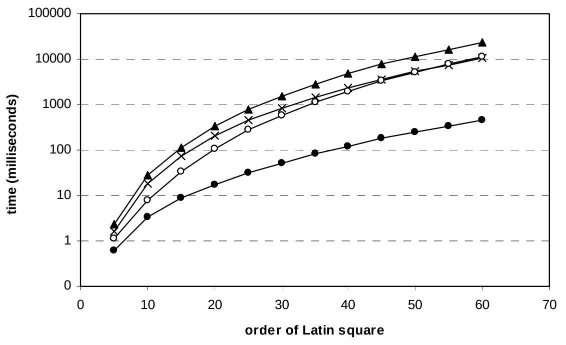

Fig. 14. The time (in milliseconds, a logarithmic scale) to generate a quasigroup problem with the filling ratio 0.6 (●: QCP-orig, ○: QCP-alldiff, ▲: QWH-orig, ✕: QWH-new).

6 Conclusions

Completion of a partial Latin square is an interesting problem whose structure is close to real-life problems [7,10]. It is also a non-trivial problem [4] whose solving requires sophisticated techniques [5,8]. Finally, it is a problem whose instances can be generated randomly as a Quasigroup Completion Problem (QCP) [6] or Quasigroups With Holes (QWH) [1]. These features make the completion of partial Latin squares an ideal candidate for benchmarking constraint satisfaction techniques. In this paper, we studied the generators for both QCP and QWH and we provided detailed guidelines how to construct such generators. This alone is an important contribution because writing the generator for QWH is a non-trivial problem. Moreover, as far as we know this is the first paper in the CSP literature giving the exact description of the generator for QWH. We experimentally compared the existing generators and we proposed a reformulated version of the QWH generator that is much faster than the original generator. Even if the QWH generators are slower than the original QCP generator, their quality measured as a number of produced satisfiable instances is much higher. Hence, the QWH generators are appropriate to prepare problem instances for testing incomplete algorithms like in [2] while the QCP generators may still be useful for testing complete algorithms like in [12].

References

1. Achlioptas, D., Gomes, C., Kautz, H., Selman, B.: Generating Satisfiable Problem Instances. In Proceedings of the Seventeenth National Conference on Artificial Intelligence. AAAI Press (2000) 256–261
2. Barták, R., Rudová, H.: Limited Assignments: A New Cutoff Strategy for Incomplete Depth-First Search. In Proceedings of the 2005 ACM Symposium on Applied Computing. ACM (2005) 388–392
3. Carlsson, M., Ottosson, G., Carlson, B.: An Open-ended Finite Domain Constraint Solver. In Programming Languages: Implementations, Logics, and Programming. LNCS 1292. Springer-Verlag (1997)
4. Colbourn, C.: The Complexity of Completing Partial Latin Squares. Discrete Applied Mathematics 8 (1984) 25–30
5. Dotú, I., del Val, A., Cebrián, M.: Channeling Constraints and Value Ordering in the Quasigroup Completion Problem. In Proceedings of Eighteenth International Joint Conference on Artificial Inteligence. Morgan Kaufmann Publishers (2003) 1372–1373
6. Gomez, C., Selman, B.: Problem Structure in the Presence of Perturbations. In Proceedings of Fourteenth National Conference on Artificial Intelligence. AAAI Press (1997) 221–226
7. Gomez, C., Shmoys, D.: Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem. In Proceedings Computational Symposium on Graph Coloring and Generalizations (2002)
8. Gomez, C., Shmoys, D.: The Promise of LP to Boost CSP Techniques for Combinatorial Problems. In Proceedings CPAIOR'02 (2002) 291–305
9. Jacobson, M.T., Matthews, P.: Generating Uniformly Distributed Random Latin Squares. Journal of Combinatorial Designs 4 (1996) 405–437
10. Kumar, S.K., Russell, A., Sundaram, R.: Approximating Latin Square Extensions. Algorithmica 24 (1999) 128–138
11. MacIntyre, E., Prosser, P., Smith, B., Walsh, T.: Random Constraint Satisfaction: theory meets practice. In Principles and Practice of Constraint Programming - CP98. LNCS 1520. Springer-Verlag (1998) 325–339
12. Meseguer, P., Walsh, T.: Interleaved and Discrepancy Based Search. In Proceedings of 13th European Conference on Artificial Intelligence. Wiley (1998) 239–243
13. Régin, J.-Ch.: A filtering algorithm for constraints of difference in CSPs. In Proceedings of Twelfth National Conference on Artificial Intelligence. AAAI Press (1994) 362–367
14. Shaw, P., Stergiou, K., Walsh, T.: Arc Consistency and Quasigroup Completion. In Proceedings of the ECAI-98 workshop on non-binary constraints (1998)