J. Symbolic Computation (1996) **21**, 543–560

PSATO: a Distributed Propositional Prover and Its Application to Quasigroup Problems

HANTAO ZHANG[†], MARIA PAOLA BONACINA[‡]AND JIEH HSIANG[*§]

Dept. of Computer Science, The University of Iowa

Iowa City, IA 52242-1419, U.S.A. {*hzhang, bonacina*}*@cs.uiowa.edu*

[*] *Dept. of Computing Science and Information Engineering*

National Taiwan University, Taipei, Taiwan *hsiang@csie.ntu.edu.tw*

(Received 31 May, 1995)

We present a distributed/parallel prover for propositional satisfiability (SAT), called PSATO, for networks of workstations. PSATO is based on the sequential SAT prover SATO, which is an efficient implementation of the Davis–Putnam algorithm. The master-slave model is used for communication. A simple and effective workload balancing method distributes the workload among workstations. A key property of our method is that the concurrent processes explore disjoint portions of the search space. In this way, we use parallelism without introducing redundant search. Our approach provides solutions to the problems of (i) cumulating intermediate results of separate runs of reasoning programs; (ii) designing highly scalable parallel algorithms and (iii) supporting "fault-tolerant" distributed computing. Several dozens of open problems in the study of quasigroups have been solved using PSATO. We also show how a useful technique called the cyclic group construction has been coded in propositional logic.

1. Introduction

We are concerned with distributed implementations of constraint satisfaction algorithms on networked environments. Nowadays, there are dozens of workstations in a typical industrial or academic research laboratory. Such networked workstations represent great computational power that is often underused, especially after hours. This paper shows a way of using such resources to solve hard problems.

Constraint satisfaction has many important applications in many areas of computer science and artificial intelligence. In recent years, there has been considerable renewed interest in a special case of constraint satisfaction: propositional satisfiability (SAT) problems. For instance, many open quasigroup problems in algebra have been reportedly

[†] Supported in part by NSF under grants CCR-9504205 and CCR-9357851.
[‡] Supported in part by the National Science Foundation under grant CCR-94-08667.
[§] Supported in part by the National Science Council of the Republic of China under grant NSC 85-2213-E-002-052.

© 1996 Academic Press Limited

solved by SAT provers [e.g., Slaney et al. (1995), Zhang and Stickel (1994) and McCune (1994)]. The difficulty with using a SAT prover as a tool in artificial intelligence is that SAT is NP-complete, and this intractability is a significant obstacle in real problems. No matter how good the algorithms are, we are in constant need of more computing power, in order to solve a significative range of SAT problems in tolerable time. This justifies the importance of exploiting underused networked workstations.

Our goal is to provide a convenient and effective way of using networked workstations to attack hard constraint satisfaction problems. To achieve this goal, we will propose answers for the following issues.

1 *Accumulation of work over time*:
Since the problems are hard, the prover may have to run for many days. For a parallel prover that may need to start a process on each machine in the laboratory, it may be difficult for the experimenter to keep the prover running for so long, because machines are shared with other users, or faults may occur. Thus, we would like to find simple and effective ways to cumulate the effort of separate runs of the prover on the same problem, e.g. run the parallel prover for a certain number of hours, then suspend, and be able to continue later without repeating the work already done.
2 *Scalability*:
A parallel implementation on networked workstations should be scalable, so that the more workstations there are, the faster a problem can be solved.
3 *Fault-tolerance*:
The parallel algorithm should be fault-tolerant, meaning that the failure of one workstation or the interruption of the network should cause only minimal damage.

Most existing reasoning systems require continuous running time. If a solution is not found within the allotted time, the execution is aborted and all the effort is wasted. For instance, an efficient parallel SAT prover is reported in Boehm and Speckenmeyer (1994) but the machine they used is a parallel computer with continuous run time. We feel that our solutions can be used not only for solving SAT problems, but also for other constraint satisfaction problems. This is a realistic expectation, since many constraint satisfaction problems can be naturally converted into SAT problems.

We chose the Davis–Putnam algorithm (Davis and Putnam, 1960) as our basic tool, because this method has been a major practical method for solving SAT problems. The method is based on unit-propagation (i.e., unit-resolution and unit-subsumption) and case analysis (splitting). We already had an efficient sequential implementation of the Davis–Putnam algorithm, called SATO[†] (SAtisfiability Testing Optimized) (Zhang, 1993; Zhang and Stickel, 1994). This paper describes a parallel implementation of SATO, called PSATO.

We have been experimenting with our department's workstations (HPs, Sparcs, SIGs and IBM RS6000s) connected by a local area network (LAN). A public domain distributed language called P4 is currently installed on this network. P4, developed at Argonne National Laboratory, provides a C library for programming a variety of parallel machines (Butler and Lusk, 1992). Although public domain languages other than P4 are available, it appears that a different choice would not make much difference, since

[†] SATO is written in C and is available by anonymous ftp from cs.uiowa.edu.

our implementation intends to have the least dependency on communication over the network.

One of the major motivations for developing our programs is to attack open quasigroup problems in algebra (Bennett and Zhu, 1992). The theory of quasigroup contains many hard computational problems. The usefulness of advanced automated reasoning techniques in attacking these quasigroup problems was successfully demonstrated in Zhang (1990), Fujita et al. (1993), Slaney et al. (1995) and Zhang and Stickel (1994). Several open problems were solved using the approach described in this paper; other reasoning systems have not been able to reproduce many of these results.

Because the SAT problem is the first known NP-complete problem, it is relatively easy to transform other NP problems in mathematics, computer science and electrical engineering into a SAT problem. The SAT problem is known to be difficult to solve in theory. However, contrary to the common perception that transforming a problem into the SAT problem will not make the problem easier to solve, our experiences show that many problems can be solved more effectively by a SAT problem solver than by a special program. Needless to say, transforming a problem into a SAT problem is much easier than writing a special program. In this paper, we show as an example that transforming the so-called cyclic group construction technique into a SAT problem is an efficient way of solving the existence problem of quasigroups satisfying certain constraints.

This paper is organized as follows: the next section introduces SATO and provides a solution on how to cumulate the search effort of SATO based on the notion of *guiding paths*. Section 3 presents the master–slave model used for PSATO. Section 4 demonstrates how PSATO is used to solve open quasigroup problems. The last section summarizes the paper. Preliminary results of this paper appeared in Zhang and Bonacina (1994) and Zhang and Hsiang (1994).

2. SATO: A SAT Prover

Our parallel implementation of a SAT prover, PSATO, is based on a sequential SAT prover called SATO (SAtisfiability Testing Optimized) (Zhang, 1993), which is an efficient implementation of the Davis–Putnam algorithm. In SATO, the *trie* data structure (i.e., discrimination trees) is used for representing clauses, and a sublinear decidability algorithm for Horn theory is used for unit propagation (Zhang and Stickel, 1994). In order to understand the parallel implementation, we first review the Davis–Putnam algorithm; we then show how to cumulate search based on the notion of guiding paths of the Davis–Putnam algorithm.

2.1. THE DAVIS–PUTNAM ALGORITHM

We assume that the reader is familiar with propositional logic and related concepts: given a language of propositional variables, a *literal* is a propositional variable (positive literal) or its negation (negative literal), a *clause* is a disjunction of literals, and a *model* for a set of clauses is an assignment of truth values to the propositional variables that makes all clauses in the set true. The Davis–Putnam algorithm accepts a set of propositional clauses and returns true (or models) if and only if the input clauses are satisfiable.

A simple algorithm based on `unit-propagation` (i.e., a process of repeated unit resolution and unit subsumption) and `case-splitting` is shown in Figure 1. The algorithm

```
function Satisfiable ( set S ) return boolean
    repeat /* unit propagation */
        for each unit clause $L$ in $S$ do
            delete $(L \vee Q)$ from $S$ /* unit-subsumption */
            delete $\overline{L}$ from $(\overline{L} \vee Q) \in S$ /* unit-resolution */
        od
        if $S$ is empty then return $TRUE$
        else if the null clause is in $S$ then return $FALSE$
        fi
    until no further changes result
    choose a literal $L$ occurring in $S$ /* case-splitting */
    if Satisfiable ( $S \cup \{L\}$ ) then return $TRUE$
    else if Satisfiable ( $S \cup \{\overline{L}\}$ ) then return $TRUE$
    else return $FALSE$
    fi
end function
```

Figure 1. A Simple Davis–Putnam Algorithm.

returns *TRUE*, if a model is found, or *FALSE*, if the input clauses are unsatisfiable. It is well-known that this algorithm is sound and complete for propositional problems.

In SATO, `unit-propagation` is done by using a new algorithm for the satisfiability of Horn theory which avoids unit-subsumption. It has been proved in Zhang and Stickel (1994) that the complexity of the new algorithm is linear in the number of literals which are false at the end of `unit-propagation`.

Note that in `case-splitting`, the clause set S is used twice in the two recursive calls of `Satisfiable`. In SATO, the trie data structure is used in such a way that the recursion in `Satisfiable` is implemented without requiring new memory, e.g. without duplicating S. This is also significant for the performance of the algorithm, because allocating memory frequently is relatively slow.

Naturally, one important place where heuristics may be inserted is in the choice of a literal for splitting. It is well-known that different splitting rules make the performance of the Davis–Putnam algorithm different by a magnitude of several orders. Splitting rules have been extensively studied in a recent paper (Hooker and Vinay, 1995). SATO provides several popular splitting rules. However, in our study of quasigroup problems, one rule seems better than the others: *choose one literal in one of the shortest positive clauses* (a positive clause is a clause where all the literals are positive).

2.2. CUMULATING SEARCH IN SATO

By "cumulating search" in solving a SAT problem, we mean that we may visit the search space of the SAT problem one portion at a time without overlapping. There is no standard definition of the search space of a SAT problem. For example, one may consider the set of all assignments of the propositional variables appearing in the input as its search space. In this paper, we consider the search tree of `Satisfiable` for a given SAT problem as its search space. The search tree of `Satisfiable` is a binary tree, which represents the relation of recursive calls of `Satisfiable` in the usual way. That is, each node represents a call to `Satisfiable`. Obviously, the search tree of the Davis–Putnam algorithm for a given input depends on the splitting rule used by the algorithm. Once the splitting rule is fixed, the search tree is fixed. Other than some general bounds, we cannot know beforehand exactly how large a search tree is.

Given any two nodes a and b in a search tree, there is a link from a to b with label $\langle L, \delta \rangle$ if and only if a calls b with input $S \cup \{L\}$, where S is the input to a, and δ is either 1, if b is the first child of a, or 0, if b is the second one. In this way, a path from the root to any node can be represented by

$$(\langle L_1, \delta_1 \rangle \langle L_2, \delta_2 \rangle \cdots \langle L_k, \delta_k \rangle)$$

where $\langle L_i, \delta_i \rangle$, $1 \leq i \leq k$, are the labels of the edges in the path. We say $\langle L, \delta \rangle$ is *open*, if δ is 1, or *closed*, if δ is 0. The algorithm visits the tree by depth-first search, where open pairs are backtracking points. It returns *TRUE*, if at least one leaf of the search tree returns *TRUE*, or *FALSE*, if all the leaves of the search tree (which is finite) return *FALSE*.

`Satisfiable` may be forced to stop prematurely at a node of the search tree. This may happen for different reasons, including that the allotted time has expired or the program has run out of memory. Whenever this happens, the path from the root to that node, called *guiding path*, provides very valuable information about the nested calls of `Satisfiable`. We can use guiding paths to avoid repeated search when `Satisfiable` is called the next time with the same input. For instance, if `Satisfiable` halts with the guiding path $(\langle x_1, 1 \rangle \, \langle x_5, 0 \rangle \langle x_3, 0 \rangle)$, then when `Satisfiable` is called again with the same input, there is no need to go through paths such as $(\langle x_1, 1 \rangle \langle x_5, 1 \rangle \cdots)$ in the search tree.

This idea of using guiding paths is easy to implement in the Davis–Putnam algorithm: we design a Davis–Putnam algorithm that accepts as input, together with the input clauses, a guiding path of the search tree. If the given path is empty, the Davis–Putnam algorithm is unchanged. If the path is not empty, then when choosing a literal for splitting, instead of choosing one according to any given splitting rule, we simply extract one pair from the guiding path. If the pair is closed, only the recursive call prescribed by the pair is made, with no backtracking at this point. We call this version of the Davis–Putnam algorithm, which has been implemented in SATO, `Satisfiable-guided`.

`Satisfiable-guided` allows us to cumulate the search done in discrete time segments by the Davis–Putnam algorithm: when the search is interrupted, the guiding path produced so far is saved and will be given to the algorithm at the next run. Our experiments show that for a typical hard SAT problem, which requires several days of running time, the length of the guiding path after 8 h of computation is between 70 and 200, but the number of branches (or leaves) of the search tree visited during the same period of time is over a million. In other words, the cost of revisiting a guiding path is so small that it can be ignored. Furthermore, it is much faster to pop a pair from the path in `Satisfiable-guided` than to choose a literal by the splitting rule. `Satisfiable-guided` can also be used to simplify the implementation of the incremental Davis–Putnam algorithm described in Hooker (1993), where the input clause set grows each time the Davis–Putnam algorithm is called.

In short, the notion of guiding paths provides us with a simple and effective way to cumulate the search of the Davis–Putnam algorithm, so that a hard SAT problem can be solved in discrete time segments. In the next section, we shall show how to use guiding paths to split jobs effectively in a parallel SAT prover.

3. The Master–Slave Model

PSATO is organized according to the master–slave model of distributed computation. One processor is designed as *master* and the remaining processors are *slaves*. The slaves execute in parallel `Satisfiable-guided`, and the master takes care of partitioning the work among the slaves. All communication takes place between the master and the slaves. More specifically, the master sends a job to each slave. In PSATO, a *job* is a pair (S, P), where S is the set of input clauses and P is a guiding path for `Satisfiable-guided`. Since a hard SAT problem may contain up to several millions of clauses, S is either the name of a file (containing the input clauses) or a list of parameters for a clause generator. Each slave executes `Satisfiable-guided` on the job it has received, and works autonomously until it halts with a result or it is forced to interrupt. In the first case it reports to the master the result ($TRUE$ or $FALSE$), and in the second case it reports a new guiding path. We mention here the main advantages of this approach.

1 Balancing the workload between processors is considered a very difficult part of implementing a SAT prover on parallel machines. However, for the special computing environment we consider here, that is, a set of loosely connected workstations available at discrete time periods and of uneven computing power, we have a simple and effective solution which only involves the master.

2 The network communication may be slow, crowded or unreliable, especially when the communication is through telephone lines. Therefore, we wish to minimize the extent to which the performance of the network affects the performance of our program. Symmetrically, we do not want our program to deteriorate the performance of the network. Our approach to parallelization reduces the communication to a minimum because messages are rare (each slave works autonomously on its assignment, and can continue even if the network breaks down) and small (jobs are encoded compactly in guiding paths).

3 The master–slave model is simple and easy to implement. It is flexible, since it allows the user to add any number of slaves (i.e., workstations) during the experiments. Moreover, this model allows very high scalability, that is, the more processors are used, the faster a solution can be found. This is because we are able to divide the search space of a SAT problem into mutually disjoint portions and each portion is visited only once.

In the following, we discuss these points in detail, with supporting experimental results.

3.1. THE WORK OF THE MASTER

As stated above, the master's responsibility is to manage all the slaves. That is, it

(a) broadcasts general information, including the choice of splitting rule, the allotted time, etc., to all slaves;
(b) prepares the jobs and distribute them to the slaves, in such a way as to balance the workload;
(c) receives reports from the slaves and
(d) tells the slaves to halt.

Among these duties, (b) is the most important; the others are straightforward. Balancing the workload among the slaves is a delicate task because, on one hand, the slaves should not be idle, and on the other hand, the workload balancing should take as little computing time as possible. This is not a trivial problem, because (i) there is no reliable estimate of the complexity of a SAT problem, and (ii) the power of each slave is not known exactly beforehand (because other users may be using the same workstation).

There are many ways to generate multiple guiding paths for a given SAT problem, such that the guiding paths lead the concurrent instances of `Satisfiable-guided` to search mutually disjoint portions of the search tree. For instance, we may randomly choose two propositional variables, x and y, and generate four guiding paths:

$$(\langle x,0\rangle\langle y,0\rangle),\ (\langle \overline{x},0\rangle\langle y,0\rangle),\ (\langle x,0\rangle\langle \overline{y},0\rangle),\ (\langle \overline{x},0\rangle\langle \overline{y},0\rangle).$$

In general, if we select k variables, we may obtain 2^k guiding paths. However, this way of dividing the search space is similar to choosing randomly a variable for splitting in a sequential SAT prover. This random splitting rule is known to have poor performance for many SAT problems. Hence, this is not a good way to balance the workload among the slaves.

In the following, we describe a simple way of balancing the workload, with the property that the search space of a SAT problem is divided into the same portions as would have been done by the splitting rule for a sequential SAT prover. We assume that a (good) splitting rule for a sequential SAT prover is given, and that we have a guiding path P, generated by the sequential SAT prover or by a slave in a previous run of PSATO.

DEFINITION 3.1. Given a guiding path P

$$P = (\langle L_1,0\rangle \cdots \langle L_i,1\rangle \cdots \langle L_k,\delta_k\rangle),$$

where $\langle L_i,1\rangle$ is the first open pair in P (counting from left to right), the two *splits* of P are P_1 and P_2:

$$\begin{array}{rcl} P_1 &=& (\langle L_1,0\rangle,\ldots,\langle \overline{L_i},0\rangle), \\ P_2 &=& (\langle L_1,0\rangle,\ldots,\langle L_i,0\rangle,\ldots,\langle L_k,\delta_k\rangle). \end{array}$$

LEMMA 3.2. *Let P_1 and P_2 be the two splits of P, and S be a set of input clauses. The search space of* `Satisfiable-guided`(S,P) *is the union of those of* `Satisfiable-guided`(S,P_1) *and* `Satisfiable-guided`(S,P_2).

PROOF. Trivial.

In our implementation, the master always maintains a list of guiding paths, according to the rule that the number of guiding paths be about 10% higher or 4 more than the number of slaves. If it is smaller than that, we split one path according to Definition 3.1[†]. We control the number of paths in this way, because we do not want to spend resources unnecessarily in maintaining a large number of them. The list of guiding paths is sorted by increasing length; paths of equal length are sorted by increasing number of open pairs, and remaining ties are broken arbitrarily.

[†] If there are no open pairs in any path, then splitting is not possible and at least one slave will be idle. Our experience is that this is often a sign that the search space of a SAT problem is about to be exhausted.

The master assigns the next unassigned path in the list to the next idle slave. If a slave has found a model, the master tells all the slaves to halt and then it terminates. If a slave has found that the clause set assigned to it is unsatisfiable, the master assigns another path to that slave. When the allowed time expires, the master sends the halt signal to all the slaves and, at the same time, collects new paths from each slave. The new paths are saved together with the unassigned paths for the next run.

3.2. THE WORK OF A SLAVE

Upon receiving a job, $\langle S, P\rangle$, and an allotted time from the master, each slave runs `Satisfiable-guided`(S, P), until one of the following events occurs:

1 `Satisfiable-guided`(S, P) has found a model (return *TRUE*) or has completed the search (return *FALSE*). In this case, the slave reports the result to the master.
2 The slave has received a halt signal from the master, or the allotted time (wall-clock time) has run out. The slave interrupts `Satisfiable-guided`(S, P) and reports the new guiding path (i.e., the information about the nested calls of `Satisfiable-guided` performed so far) to the master.

Allotted run time for each slave is managed by the master. If the master dies or hangs, it will not be able to halt the slaves. Thus, the slaves must be able to halt by themselves by checking their allotted run time. This provision allowed us to overcome a problem with the version of P4 available at the time of the implementation of PSATO: if the master tries to communicate with a dead slave (there are many factors to bring a slave to death, such as physical failure, insufficient memory, etc.), the master hangs, whereas all the other slaves continue working. Awareness of these potential problems with distributed implementations is another reason why we designed our program to be as independent on the network as possible. Our implementation achieves some additional fault-tolerance by establishing that each slave saves in a file or sends by electronic mail to some address a copy of all the information that the slaves sends to the master. In this way, even when the master is dead, the effort of all the slaves will not be wasted, because all the information generated by those slaves can be retrieved.

3.3. PERFORMANCE EVALUATION

The master–slave model described above has been implemented in PSATO using P4 as the communication tool and SATO as a sequential SAT prover. Because a fine-grained algorithmic complexity analysis of PSATO appears impossible, we conducted some testing in the hope that it may shed some light on the issue. A more extensive empirical study would be more desirable but very expensive, since it would require finding a representative set of domain examples and running each parameter scheme several times (once for each setting) on each problem.

Because of the simplicity of the workload balancing method – splitting guiding paths – we do not expect our method to perform regularly well on tests made of a single run. On the other hand, since the computing environment under discussion only allows separated run times, it is more realistic to test the model in this environment. In fact, we do not expect a fancy balancing algorithm to do better, because the powers of the workstations

Table 1. Experiment of PSATO on random 3-SAT unsatisfiable problems.

#V	#P	Wall Clock	Total Time	Speed–up	Overhead
	1	22.2	22.2	–	–
100	5	7.9	24.4	2.81	0.10
	20	3.6	26.0	6.17	0.17
	1	1082.5	1082.5	–	–
150	5	237.9	1169.1	4.55	0.08
	20	60.7	1212.4	17.83	0.12
	1	53346.7	53346.7	–	–
200	5	10777.0	54947.1	4.95	0.05
	20	2899.3	58793.4	18.40	0.07

are different and cannot be known *a priori* (since other programs may be running at the same time).

We have chosen as test cases random unsatisfiable 3-SAT problems; for satisfiable 3-SAT problems, the performance of PSATO varies dramatically from case to case. Each problem consists of m 3-literals clauses over n variables with ratio $m/n = 4.25$. These have been considered as hard SAT problems (Goldberg, 1979). For $n = 100$ (and $m = 425$), we tested a sample of 50 cases on 1, 5 and 20 workstations, respectively. We did the same for $n = 150$. For $n = 200$, we tested only a sample of 20 cases which can be finished in 24 h on a single machine.

In these experiments with PSATO, the parallel execution is done as follows: first, the master assigns the entire job to one slave with 1 sec of allotted time. After this time expires, the master splits the guiding path obtained from that slave into more paths and distributes them among the slaves, with 1 min of allotted time for each slave. After that, the allotted time will be 1 h for the rest of the execution. We gave slaves small time slots, because the test problems are easy in comparison with real application problems, which require many days to solve. For hard SAT problems from real applications, we used each workstation for about 8 h a day (from midnight to 08:00hrs).

The experimental data are listed in Table 1. The 20 workstations consists of 10 HP 700 series workstations, six IBM RS6000 and four Sparc workstations. The RS6000 was chosen for the experiments with one and five machines, because it has the average computing power of the workstations available. In Table 1, #V is the number of variables, #P is the number of processors (or workstations), and all times are measured in seconds. The "wall clock" is the average time to solve each test case. The "total time" is the sum of the CPU times of all the involved machines[†]. The "speed-up" is the ratio between the wall-clock time of the sequential prover and the current wall-clock time. It is clear from the table that the harder are the test cases, the higher are the speed-up's, because for hard problems, the master has more chances to manage guiding paths and balance workloads.

We remark that speed-up is an interesting measure, but it is not the best or the only

[†] For the one workstation case, the wall-clock time is in general slightly larger than the cpu time. For simplicity, we assume they are the same.

indicator for the computing environment under discussion, because a slow workstation may contribute very little but is still counted as one machine. The "overhead" of parallel execution may provide more useful information. The "overhead" is the ratio of the extra CPU time (i.e., the current total CPU time minus the sequential CPU time) incurred by the parallel implementation versus the sequential CPU time. This includes the time spent on communication, balancing workload and re-creating data structures. Again, we found that the "overhead" becomes smaller as the difficulty of the problems increases. The data also confirm that our method has very good scalability.

4. Quasigroup Problems

In our opinion, quasigroup problems are much better benchmarks than randomly generated SAT problems for testing constraint-solving methods. Quasigroup problems are real problems, have fixed solutions and simple descriptions that are easy to communicate. More importantly, open problems in quasigroups have attracted the interest of several researchers and became a challenge for friendly competition.

The usefulness of general automated reasoning techniques to attack these quasigroup problems was successfully demonstrated in Zhang (1990), Fujita et al. (1993), Zhang (1993), Slaney et al. (1995), Zhang and Stickel (1994) and McCune (1994). To our knowledge, J. Zhang was the first to use a general reasoning program to solve an open case of the quasigroup problems. Subsequently, Fujita used MGTP, a model-generation based first-order theorem prover, and Slaney used FINDER, a program based on constraint solving, to solve several open cases. Later, Stickel, McCune, and the first author used their propositional provers to attack these problems and reported very competitive results. In particular, PSATO is able to reproduce all the results obtained by other systems.

4.1. THE SPECIFICATION OF QUASIGROUPS

A quasigroup is a cancellative finite groupoid $\langle S, * \rangle$, where S is a set and $*$ a binary operation on S. The cardinality of S, $|S|$, is called the *order* of the quasigroup. The "multiplication table" for the operation $*$ forms a Latin square indexed by S, where each row and each column is a permutation of S. Many classes of Latin squares are interesting, partly because they are very natural objects in their own right and partly because of their relationships to design theory (Bennett and Zhu, 1992).

Without loss of generality, let S be $\{0, 1, \ldots, (v-1)\}$, where v is the order of $\langle S, * \rangle$. The following clauses specify a quasigroup, or equivalently, a Latin square: for all elements $x, y, u, w \in S$,

$$x * u = y, x * w = y \Rightarrow u = w \quad \text{the left-cancellation law} \quad (4.1)$$

$$u * x = y, w * x = y \Rightarrow u = w \quad \text{the right-cancellation law} \quad (4.2)$$

$$x * y = u, x * y = w \Rightarrow u = w \quad \text{the unique-image property} \quad (4.3)$$

$$(x * y = 0) \vee \cdots \vee (x * y = (v-1)) \quad \text{the closure property.} \quad (4.4)$$

It was shown in Slaney et al. (1995) that the following two clauses are valid consequences of the above clauses and adding them into a prover may reduce the search space.

$$(x * 0 = y) \vee \cdots \vee (x * (v-1) = y) \quad \text{: the closure property (a)} \quad (4.5)$$

$$(0 * x = y) \vee \cdots \vee ((v-1) * x = y) \quad \text{: the closure property (b).} \quad (4.6)$$

Name	**Constraint**
QG1	$(x * y) * y = (x * z) * z \Rightarrow y = z$
QG2	$(y * x) * y = (z * x) * z \Rightarrow y = z$
QG3	$(x * y) * (y * x) = x$
QG4	$(x * (y * x)) * y = x$
QG5	$((x * y) * x) * x = y$
QG6	$(x * y) * y = x * (x * y)$
QG7	$((x * y) * x) * y = x$

Figure 2. Special constraints of quasigroup problems.

In Figure 2, we list the quasigroup problems given in Slaney et al. (1995). The variables x, y, z in the constraints are implicitly universally quantified over the domain $\{0, \ldots, (v-1)\}$.

In the following, QG$i(v)$ denotes a Latin square of order v that satisfies clauses (4.1)–(4.6) plus QGi for $S = \{0, \ldots, (v-1)\}$. In addition, the *idempotency* law, $x * x = x$, and the isomorphism cutting clause, $1 + x < y \Rightarrow x * 0 \neq y$, are assumed in every problem unless otherwise stated. The reader may find additional details on these problems in Fujita et al. (993) and Slaney et al. (1995).

Propositional clauses are obtained simply by instantiating the variables in clauses (4.1)–(4.6) by values in S and replacing each equality $x * y = z$ by a propositional variable $p_{x,y,z}$. To achieve this, we need to transform the constraints into a "flat" form. For example, the flat form of QG1 is

$$(t * y = u), (w * z = u), (x * y = t), (x * z = w) \Rightarrow y = z$$

and the flat form of QG5 is

$$(x * y = z), (z * x = w) \Rightarrow (w * x = y).$$

It can be shown that the two "transposes" of the above clause are also valid consequences of QG5:

$$(w * x = y), (x * y = z) \Rightarrow (z * x = w),$$
$$(z * x = w), (w * x = y) \Rightarrow (x * y = z).$$

It has been confirmed by experiments that adding these "transposes" to the input may reduce the search space. This is true at least for the problems QG3–QG7.

Among the Latins squares satisfying these constraints, we are often interested in those squares with a hole, i.e., a subsquare (which itself may be also a Latin square) of the square is missing. An *incomplete Latin square* is a Latin square with a single hole and is specified as $\langle S/X, *\rangle$, where $X \subset S$ and for any $x_1, x_2 \in X$, $x_1 * x_2$ is left undecided in the "multiplication table" of $*$, and for any $s \in S - X$ and $x \in X$, $s * x$ (or $x * s$) $\in S - X$. Let n be $|X|$ and v is $|S|$. We may consider an incomplete Latin square as one indexed by S with a subsquare of order n "missing"; if we fill a Latin square indexed by X into the hole, the result should be a Latin square of order $|S|$. Without loss of generality, the missing subsquare can be assumed to be in the bottom right corner. A necessary condition for the existence of incomplete quasigroups satisfying QG1–QG7 is that $v > 3n$.

We will denote an incomplete quasigroup of order v satisfying QGi, with a hole of size

Table 2. Performance of PSATO on some quasigroup problems.

Problem	Models	Branches	Create (sec.)	Search (sec.)
QG1(7)	8	376	1	1
(8)	16	102610	3	379
QG2(7)	14	340	1	1
(8)	2	80245	3	341
QG3(8)	18	1072	0.1	3
(9)	0	48545	0.3	157
QG4(8)	0	925	0.1	2
(9)	178	52826	0.2	168
QG5(9)	0	19	0.1	0.2
(10)	0	62	0.3	0.5
(11)	5	111	1	2
(12)	0	369	2	7
(13)	0	10764	3	224
QG6(9)	4	24	0.2	0.2
(10)	0	150	0.4	0.7
(11)	0	519	1	6
(12)	0	5728	1	92
QG7(9)	4	7	0.2	0.2
(10)	0	54	0.4	0.4
(11)	0	254	1	3
(12)	0	1281	2	22
(13)	64	27988	2	592

n, by QG$i(v,n)$. Note that every QG$i(v)$ can be considered as a QG$i(v,1)$ if there exists an element a in QG$i(v)$ such that $a*a=a$.

There is an easy way to obtain clauses for incomplete quasigroups from those of complete quasigroups.

1 Add the positive unit clauses such as $x*y=z$ whenever $\{x,y,z\}\subseteq X$.
2 Add the negative unit clauses such as $x*y\neq z$ whenever two of $\{x,y,z\}$ are in X and the other is not in X.
3 Attach the condition that x and y cannot be in X at the same time, to clauses (4.1)–(4.3).

Table 2 shows the performance of PSATO on quasigroup problems of small orders. The time (in seconds) was collected on a Sparc2 workstation[†]. "Branches" is the number of branches (or leaves) of the corresponding search tree. "Create" is the time spent on creating the internal data structure. "Search" is the time spent on search. It is clear that the complexity is exponential in the order of quasigroups. QG5(9), the first open quasigroup problem solved by J. Zhang (1990), can be considered as trivial now. Note that when PSATO runs on multiple processors, the work of creating data structure will be repeated every time a slave receives a job, thus it is part of the overhead of parallelism.

[†] PSATO is implemented in such a way that if no slaves are provided, then PSATO runs as SATO.

4.2. THE CYCLIC GROUP CONSTRUCTION

In general, the number of the propositional clauses for a given quasigroup problem is determined by the order of the quasigroup (i.e., v) and the number of distinct variables in its "flat" clauses. For instance, the number of propositional clauses obtained from QG1 and QG2 in Table 2 is $O(v^6)$ because there are six distinct variables in the flat forms of QG1 and QG2. For large v, besides the large number of clauses, the search space involved in these problems is also huge. For instance, PSATO, or any known computer programs, including those in Zhang (1990), Fujita et al. (1993), Slaney et al. (1995), Zhang and Stickel (1994) and McCune (1994), could not complete an exhaustive search when $v \geq 10$ for QG2(v). As a result, a brute-force approach based on the direct representation of quasigroups in propositional logic is not likely to succeed.

Here, we present an incomplete technique which has been developed by mathematicians over the years for finding Latin squares. It is incomplete because when the method fails, we do not know if there exists a Latin square satisfying the given constraints. We have used several other techniques, but this one is the most important because it reduces the search space most significantly. This technique is a starter-adder-type construction and has been used extensively by various authors [e.g., see Horton (1974), Hedayat and Seiden (1974) and Bennett and Zhu (1992)]. The main idea of the technique is to generate an incomplete quasigroup under an Abelian group of order $v - n$ (e.g., $(Z_{v-n}, +)$), from its first row and from the last n elements of the first column.

Suppose that L is a QG$i(v, n)$ based on S with a hole indexed by X. Let $S = G \cup X$ where $G = \{0, 1, \ldots, v - n - 1\}$ and $X = \{x_1, x_2, \ldots, x_n\}$. We will denote by $e(i, j)$ the entry in the cell (i, j) of L (i.e., $e(i, j) = i * j$). The first row is given by the vectors $\mathbf{e} = (e(0, 0), \ldots, e(0, v - n - 1))$ and $\mathbf{f} = (e(0, v - n), \ldots, e(0, v - 1))$, and the last n elements of the first column are given by the vector $\mathbf{g} = (e(v - n, 0), \ldots, e(v - 1, 0))$.

CYCLIC GROUP CONSTRUCTION 4.1. *The whole L is constructed from* $\mathbf{e}, \mathbf{f}$ *and* $\mathbf{g}$ *using the cyclic group Z_m, where $m = v - n$, as follows.*

1 *For* $0 \leq s, t < m$, $e(s + 1, t') = e'$ *where* $t' = t + 1 \pmod m$, *and* $e' = e(s, t)$ *if* $e(s, t) \in X$, *or* $e(s, t) + 1 \pmod m$, *otherwise.*
2 *For* $0 \leq s < m$, $m \leq t < v$, $e(s + 1, t) = e(s, t) + 1 \pmod m$.
3 *For* $m \leq s < v$, $0 \leq t < m$, $e(s, t + 1) = e(s, t) + 1 \pmod m$.

EXAMPLE 4.1. If $G = Z_{13}$, $X = \{x\}$, $\mathbf{e} = (0\ 12\ 9\ x\ 10\ 6\ 8\ 4\ 11\ 5\ 1\ 3\ 7)$, $\mathbf{f} = (3)$ and $\mathbf{g} = (12)$, we have QG2(14, 1). It gives us a QG2(14) when entry $e(x, x)$ is filled by x. □

EXAMPLE 4.2. If $G = Z_{15}$, $X = \emptyset$, $\mathbf{e} = (0\ 5\ 11\ 10\ 7\ 4\ 2\ 8\ 3\ 14\ 1\ 13\ 9\ 6\ 12)$, $\mathbf{f} = \mathbf{g} = \emptyset$, we have a QG2(15, 0), or QG2(15). □

The above construction can be generalized by using an arbitrary abelian group instead of the cyclic groups. A QG1(12) was found using this approach.

In order to find a new Latin square using the cyclic group construction, instead of looking for a whole square, we look for only vectors $\mathbf{e}$, $\mathbf{f}$ and $\mathbf{g}$. Obviously, the latter is much easier than the former because a square of order v has v^2 entries while the three vectors have at most $1.3v$ entries altogether (because $v > 3n$).

Table 3. The statistics of PSATO on QG2$(v,1)$ for $v = 7$ to 12.

v	Model	Clause	Branch	Search
7	6	11134	50	0.15
8	12	28883	107	0.63
9	28	67501	475	3.35
10	0	143938	1186	11.99
11	100	284036	4606	71.21
12	0	525229	14111	302.56

There are conditions that the vectors $\mathbf{e}, \mathbf{f}$ and $\mathbf{g}$ must satisfy in order to produce QG$i(v,n)$. Instead of putting the constraints on the three vectors, we simply add instances of the following clauses to the clauses of a quasigroup problem:

$$\begin{aligned} x*y=z &\Rightarrow (x+1)*(y+1)=(z+1), \\ (x+1)*(y+1)=(z+1) &\Rightarrow x*y=z, \end{aligned}$$

for all $x, y, z < m$, and

$$\begin{aligned} x*y=z &\Rightarrow (x+1)*(y+1)=z, \\ (x+1)*(y+1)=z &\Rightarrow x*y=z \end{aligned}$$

for all $x, y < m$ and $z \geq m$, where $+$ is modulo $m = v - n$. This way, we obtain a square directly from a SAT prover instead of the three vectors but this square can be reconstructed from its first row and column by the cyclic group construction. This implementation reduces substantially the search space of a quasigroup problem.

Table 3 shows the experimental results of PSATO (on a single workstation) using the cyclic group construction on the QG2$(v,1)$ problems for $v = 7$ to 12. This is, we add clauses like $x*y=z \Rightarrow (x+1)*(y+1)=(z+1)$, etc., in the input. **Model**, **Clause** and **Branch** give, respectively, the numbers of models, input clauses and leaves of the search tree. **Search** gives the time (in seconds), collected on a single HP715/50 workstation with 32 megabytes of memory, spent in searching for models. The advantage of the cyclic construction is obvious. For instance, it takes 379 sec for PSATO to complete the search of QG2(8) without the cyclic group construction but it takes only 0.63 sec when the cyclic group construction is used. However, no QG2(12) is found by the cyclic group construction even though we know it exists, because the method is incomplete.

4.3. NEWLY SOLVED OPEN CASES

Despite of the difficulty of quasigroup problems, many of them can be solved by computers. Table 4 shows some recently solved problems. Y means “Yes”, that is, a model has been found; N means “No”, that is, an exhaustive search confirms that no models exist. O means that the case remains open. PSATO is able to reproduce all the results in this table. In particular, PSATO was the first to solve QG2(14), QG2(15), QG5(14) (without idempotency), QG6(15), QG7(15), QG6(17), and many incomplete Latin squares. Except for QG2(14), QG2(15) and QG6(17), we do not know of other programs which reproduce these results.

Our program made new discoveries in every type of the quasigroup problems, QG1–QG7, listed in Fujita et al. (1993). Our program is also able to reproduce all the results reported in Zhang (1990), Fujita et al. (1993), Slaney et al. (1995) and McCune (1994).

Table 4. Open quasigroup problems recently solved by a computer prover.

v :	9	10	11	12	13	14	15	16	17	18	20	33
QG1				Y								
QG2		O		Y		Y	Y					
QG3				Y								
QG4				Y								
QG5	N	N		N	N	N	N	N	O	O		
QG6		N	N	N		N	N		Y		O	
QG7		N	N	N		N	N					O

Table 5. Quasigroup problems newly solved by PSATO.

QGi	(v) or (v, n)
QG1	(12), (16, 2), (17, 2), (19, 2), (20, 3)
QG2	(14), (15), (12, 2), (14, 2), (16, 2), (17, 2), (15, 3), (17, 3), (18, 4)
QG3	(46, 15)
QG4	(14, 2), (15, 2)
QG5	(14)[without idempotency], (16)
QG6	(15), (17)
QG7	(15)

In Table 5 we list the open problems of Bennett and Zhu (1992) which were solved first by our prover. When a case is solved by the cyclic group construction, instead of presenting the newly found Latin squares, we often list the vectors **e**, **f** and **g**, which can be used to construct the whole square by the cyclic group construction.

Among these cases, QG4(14, 2) and QG4(15, 2), given in Figure 4 in the Appendix, were found without the help of the cyclic group construction. The cases for QG5(14), QG5(16), QG6(15) and QG7(15) are negative, i.e., no such squares exist. The case of QG5(16) has been proved independently also by Hasegawa's team in Japan. The other cases are positive and the squares are obtained by the cyclic group construction. The vectors for QG2(14) and QG2(15) are given in Examples 4.1 and 4.2, respectively. The other involved vectors are listed in Figure 3 in the Appendix. Note that using his propositional prover, McCune found a QG6(17) without the cyclic group construction (McCune, 1994).

One of the reasons why problems like QG5(14) (without idempotency), QG6(15) and QG7(15) are so hard is that they are unsatisfiable SAT problems. An exhaustive search is necessary to establish unsatisfiability. Since few heuristic methods can help to cut the search space, this requires an enormous amount of computing time. From Table 6, it is clear that the use of distributed programs on networked workstations is indispensable to our success. For instance, it would require approximately 240 days of continuous running on a single workstation to solve QG5(14) (without idempotency).

We observed in our experiments on quasigroups that, if a problem is unsatisfiable, guiding paths provide a good empirical measure of the complexity of the problem. Assume that the guiding path is obtained after the first work day, and n is the number

Table 6. Performance data of PSATO on hard quasigroup problems.
#P = number of machines. A work day equals 8 h. (*) means no idempotency.

Prob.	#P	Workdays	P-Measure
QG5(14*)	20	35	11
QG5(16)	20	4	5
QG6(15)	20	8	8
QG7(15)	20	5	6
QG6(17)	8	2	–
QG4(14, 2)	10	7	–
QG4(15, 2)	20	30	–

of consecutive open pairs in the beginning of this path. We call this n the *P-measure* of the given problem. Then this problem will need approximately $O(2^n)$ work days to complete (on a single workstation). For instance, the P-measure of QG6(15) is 8 and we needed 160 work days to solve it. This empirical rule does not apply to satisfiable SAT problems. P-measures of all the problems in Table 6 are given in the last column. Note that the P-measure of QG2(10) is 25. If this problem is unsatisfiable, it is unlikely that our program may confirm it, because it would take approximately 2^{25} work days to solve QG2(10).

5. Conclusions

We presented PSATO, a master–slave distributed SAT prover, for networked workstations. Each slave process executes a modified version of the sequential SAT prover SATO. The master process decomposes the given problem among the slaves, in such a way that the slaves explore non-overlapping portions of the search space in discrete time segments. In this way, we are able to exploit parallelism, without incurring in the overhead of redundant search, e.g. parallel processes searching the same portion of the search space.

Our parallel program is as easy to use as a sequential program. For instance, as the experiments may need to run for several days, we may set our program to work each night from midnight to 8am, and each morning the output of the previous run is checked and the task for the following night is prepared.[†] This experience shows that it is possible to utilize the unused computing power of networked workstations after hours to attack computationally intensive problems.

We were able to use our program to solve several open problems in the study of quasigroups, thus making contributions to a community other than theorem proving. We showed how to encode the so-called cyclic group construction in propositional logic. This construction has been implemented by many mathematicians as a special purpose program for finding many Latin squares. Judging from the success of our propositional prover, it is clear that transforming a problem into the propositional logic is an effective

[†] The "`at`" facility of Unix is used so that the program does not need to be started manually at night.

way of solving the problem, at least in the case of quasigroup problems. Quasigroup problems have been attacked by several kinds of general theorem provers. We are interested to know how the cyclic group construction can be used in these provers, especially in constraint-satisfaction-based provers such as FINDER.

In this study, we have provided solutions to the problems of (i) cumulating intermediate results of separated runs of reasoning programs; (ii) designing highly scalable parallel algorithms and (iii) supporting "fault-tolerant" distributed computing. We believe that the same approach to parallelization can be generalized at least to constraint satisfaction techniques. A similar approach might be considered for general theorem proving as well, although partioning the search space in non-overlapping portions is a much more difficult problem in general theorem proving.

Acknowledgements

We thank Bill McCune for providing us with the P4 scheme code of the master–slave model. We thank Rusty Lusk for helping us use P4, and Jian Zhang for his assistance in implementing the cyclic group construction.

References

Bennett, F., Zhu, L. (1992) Conjugate-orthogonal Latin squares and related structures. Dinitz, J., D. Stinson, D. (eds), *Contemporary Design Theory: A Collection of Surveys.*

Boehm, M., Speckenmeyer, E. (1994) A fast parallel SAT-solver - efficient workload balancing. Presented at the *Third International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, Florida.

Butler, R., Lusk, E. L. (1992) User's Guide to the p4 Programming System. Technical Report ANL-92/17, Mathematics and Computer Science Division, Argonne National Laboratory.

Chang, C. L., Lee, R. C.-T. (1973) *Symbolic Logic and Mechanical Theorem Proving.* New York, Academic Press.

Davis, M., Putnam, H. (1960) A computing procedure for quantification theory. *J. ACM*, **7**, 201–215.

Fujita, M., Slaney, J., Bennett, F. (1993) Automatic Generation of Some Results in Finite Algebra. *Proc. Int. Joint Conference on Artificial Intelligence.*

Goldberg, A. (1979) Average case complexity of the satisfiability problem. *Proc. Fourth Workshop on Automated Deduction*, 1–6.

Hedayat, A., Seiden, E. (1974) On the theory and application of sum composition of Latin squares and orthogonal Latin squares. *Pacific Journal of Math.*, **54**, 85–93.

Hooker, J. N. (1993) Solving the incremental satisfiability problem. *J. Logic Programming*, **15**, 177–186.

Hooker, J. N., Vinay, V. (1995) Branching rules for satisfiability. *J. Automated Reasoning*, **15**:(3), 359–383.

Horton, J. D. (1974) Sub-Latin squares and incomplete orthogonal arrays. *J. Combin. Theory A*, **16**, 23–33.

McCune, W. W., (1994) A Davis–Putnam program and its application to finite first-order model search: quasigroup existence problems. Preprint, Division of MCS, Argonne National Laboratory.

Slaney, J., (1992) FINDER, Finite Domain Enumerator: Version 2.0 Notes and Guide. Technical report TR-ARP-1/92, Automated Reasoning Project, Australian National University.

Slaney, J., Fujita, M., Stickel, M., (1995) Automated reasoning and exhaustive search: Quasigroup existence problems. *Computers and Mathematics with Applications*, **29**, 115–132.

Zhang, H., (1993) SATO: A decision procedure for propositional logic. *Association for Automated Reasoning Newsletter*, **22**, 1–3.

Zhang, H., Bonacina, M. P. (1994) Cumulating search in a distributed computing environment: A case study in parallel satisfiability. Hong, H. (ed.), *Proc. of the First Int. Symp. on Parallel Symbolic Computation*, Linz, Austria, World Scientific, Lecture Notes Series in Computing, **5**, 422–431.

Zhang, H., Hsiang, J., (1994) Solving open quasigroup problems by propositional reasoning. *Proc. of International Computer Symp.*, Hsinchu, Taiwan.

Zhang, H., Stickel, M. (1994) Implementing the Davis–Putnam algorithm by tries. Technical Report, Dept. of Computer Science, The University of Iowa.

Zhang, J., (1990) Search for idempotent models of quasigroup identities. Preprint, Institute of Software, Academia Sinica, Beijing.

A. Appendix

Problem	**e, f, g**
QG1(16, 2)	(0 4 9 12 8 x_1 3 13 10 5 11 2 x_2 7), (1 6), (12 13)
QG1(17, 2)	(0 6 14 10 x_2 11 1 8 x_1 12 4 13 5 9 3), (2 7), (13 14)
QG1(19, 2)	(0 14 16 11 8 7 12 2 x_2 3 15 4 13 5 x_1 1 6), (9 10), (15 16)
QG1(20, 3)	(0 12 14 16 7 10 13 x_1 9 2 1 x_2 x_3 15 6 4 3), (5 8 11), (14 15 16)
QG2(12, 2)	(0 x_2 3 6 1 10 x_1 2 11 5), (4 7), (4 9)
QG2(14, 2)	(0 x_2 9 11 7 x_1 10 8 2 6 3 1), (4 5), (10 11)
QG2(16, 2)	(0 8 x_1 5 13 10 12 11 9 6 4 7 x_2 2), (1 3), (12 13)
QG2(17, 2)	(0 7 12 6 9 13 x_1 1 4 11 2 8 x_2 14 3), (5 10), (13 14)
QG2(15, 3)	(0 x_1 10 6 5 9 11 x_3 3 x_2 4 1), (2 7 8), (9 10 11)
QG2(17, 3)	(0 7 12 11 x_3 x_2 8 10 9 13 1 6 5 x_1), (2 3 4), (11 12 13)
QG2(18, 4)	(0 x_2 x_1 5 11 x_3 10 12 9 x_4 13 3 6 8), (1 2 4 7), (10 11 12 13)
QG3(46, 15)	(0 x_6 22 2 x_7 24 4 x_8 26 6 x_9 28 8 x_{10} 30 10 x_{11} x_1 12 x_{12} x_2 14 x_{13} x_3 16 x_{14} x_4 18 x_{15} x_5 20), (1 3 5 7 9 11 13 15 17 19 21 23 25 27 29), (15 14 13 12 11 10 9 8 7 6 5 4 3 2 1)
QG7(17, 0)	(0 13 9 12 1 3 7 11 2 15 6 10 14 16 5 8 4), ( ), ( )

Figure 3. Vectors for the cyclic group construction.

```
* | 0 1 2 3 4 5 6 7 8 9 a b c d        * | 0 1 2 3 4 5 6 7 8 9 a b c d e
--+----------------------------        --+------------------------------
0 | 0 a 6 7 1 b d 9 5 c 2 4 8 3        0 | 0 6 d e c a 5 4 2 3 7 8 b 9 1
1 | 3 1 0 2 8 6 b d 9 a 4 c 5 7        1 | c 1 e 7 8 3 9 5 d 6 b a 4 2 0
2 | 4 c 2 6 b 8 3 a 7 0 d 5 9 1        2 | a 0 2 b 7 e 4 8 9 d 5 c 6 1 3
3 | 2 b d 3 5 0 8 1 c 6 7 9 4 a        3 | 1 c 6 3 e 9 b 0 5 4 8 7 d a 2
4 | 1 2 5 9 4 7 a 8 b 3 c d 0 6        4 | d b a 2 4 c 1 e 0 7 3 6 9 8 5
5 | d 8 1 a 6 5 c 3 4 2 9 7 b 0        5 | b 8 3 0 1 5 e d 6 a c 9 2 7 4
6 | 9 d 7 0 3 c 6 b 2 4 5 a 1 8        6 | 8 a b d 9 4 6 c 3 0 e 2 1 5 7
7 | 5 0 c 4 a 3 1 7 d b 6 8 2 9        7 | 9 d 0 a 5 6 2 7 e b 4 1 3 c 8
8 | 6 9 3 b 2 d 7 c 8 5 0 1 a 4        8 | 7 2 c 1 b 0 a 6 8 e d 4 5 3 9
9 | a 7 b c d 1 4 2 3 9 8 0 6 5        9 | 5 4 1 c 0 2 d 3 7 9 6 e 8 b a
a | c 6 9 8 0 4 2 5 1 d a 3 7 b        a | 3 7 9 6 d 8 c 2 4 1 a 5 e 0 b
b | 7 4 a d c 9 5 0 6 8 1 b 3 2        b | 4 e 5 9 2 d 3 a 1 8 0 b 7 6 c
c | b 5 8 1 9 2 0 4 a 7 3 6            c | e 5 7 8 3 b 0 1 a 2 9 d c 4 6
d | 8 3 4 5 7 a 9 6 0 1 b 2            d | 6 9 8 4 a 1 7 b c 5 2 3 0
                                       e | 2 3 4 5 6 7 8 9 b c 1 0 a
```

Figure 4. A QG4(14, 2) and a QG4(15, 2).