A Davis-Putnam Program and its Application to Finite First-Order Model Search: Quasigroup Existence Problems*

William McCune

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439-4844, U.S.A.
e-mail: mccune@mcs.anl.gov
phone: 708-252-3065

DRAFT May 9, 1994

Abstract

This document describes the implementation and use of a Davis-Putnam procedure for the propositional satisfiability problem. It also describes code that takes statements in first-order logic with equality and a domain size n searches for models of size n. The first-order model-searching code transforms the statements into set of propositional clauses such that the first-order statements have a model of size n if and only if the propositional clauses are satisfiable. The propositional set is then given to the Davis-Putnam code; any propositional models that are found can be translated to models of the first-order statements. The first-order model-searching program accepts statements only in a flattened relational clause form without function symbols. Additional code was written to take input statements in the language of OTTER 3.0 [5] and produce the flattened relational form. The program was successfully applied to several open questions on the existence of orthogonal quasigroups.

1 The Davis-Putnam Procedure

The Davis-Putnam procedure is widely regarded as the best method for deciding the satisfiability of a set of propositional clauses. I'll assume that the reader is familiar with it. I list here some features of our implementation.

1. There are no checks for pure literals. Experience has shown that is usually more expensive than its worth.
2. Deletion of subsumed clauses is optional. Again, experience has shown that it is too expensive.
3. The variable selected for splitting is the always first literal of the first, shortest positive clause.

*This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

1.1 Implementation

The data structures for clauses and for propositional variables are similar to the ones Mark Stickel uses in LDPP [8].

Variables are integers ≥ 1. Associated with the set of variables is an array, indexed by the variables, of variable structures. Each variable structure contains the following fields.

- `value`. The current value (true, false, or unassigned) of the variable.
- `enqueued_value`. To speed the bottleneck operation of unit propagation (see below).
- `pos_occ`. A list of pointers to clauses that contain the variable in a positive literal.
- `neg_occ`. A list of pointers to clauses that contain the variable in a negative literal.

Each clause contains the following fields.

- `pos`. A list of variables representing positive literals.
- `neg`. A list of variables representing negative literals.
- `active_pos`. The number of positive literals that have not been resolved away.
- `active_neg`. The number of negative literals that have not been resolved away.
- `subsumer`. If the clause has been inactivated by subsumption, this field is set to the responsible variable.

Assignment. When a variable is assigned a value, say true, by splitting or during unit propagation, unit resolution is performed by traversing the `neg_occ` list of the variable: for each clause that has not been subsumed, the `active_neg` field is simply decremented by 1. If `active_pos+active_neg` becomes 0, the empty clause has been found and backtracking occurs. If `active_pos+active_neg` becomes 1, the new unit clause is queued for unit propagation. In addition, if subsumption is enabled, (back) subsumption is performed by traversing the `pos_occ` list of the variable: for each clause that is not already subsumed, the `subsumer` field is set to the variable. Variables must be unassigned during backtracking, and the process is essentially the reverse of assignment.

Unit Propagation. A split causes an assignment. The unit propagation queue is then processed (causing further assignments and possibly more units to be queued) until empty or the empty clause is found. Each split typically causes many assignments, so unit propagation must be done efficiently. To avoid duplicates in the queue, and to detect the empty clause during the enqueue operation rather than during assignment, we set the field `enqueued_value` of the variable when the corresponding literal is enqueued. That way we can quickly tell whether a literal or its complement is already in the queue.

Unit Preprocessing. If the set of input clauses contains any units, unit propagation is applied. During assignment, back subumption is always applied, because assignments made during this phase are never undone.

Selecting Variables for Splitting. The variable selected for splitting is the first literal in the first, shortest nonsubsumed positive clause. After the unit preprocessing, pointers to all of the non-Horn clauses (i.e., clauses with 2 or more positive literals) are collected into a list. To select a variable for splitting, the list is simply traversed. Subsumed clauses must be ignored; if subsumption is enabled, the subsumer field is checked; otherwise the clause is scanned for a literal with value true. (If all clauses in the list are subsumed, a model has been found.)

1.2 Pigeonhole Problems

The pigeonhole problems are a set of artificial propositional problems that are used to test the efficiency of propositional theorem provers. See the sample input files that come with ANL-DP for examples. Table 1 lists the performance of ANL-DP on several instances of the pigeonhole problems. The jobs were run on a SPARC 2.

	Branches	Seconds
7 pigeons, 6 holes	719	.15
8 pigeons, 7 holes	5039	1.14
9 pigeons, 8 holes	40319	9.16
10 pigeons, 9 holes	362879	88.43
11 pigeons, 10 holes	3628799	916.62

Table 1: ANL-DP on the Pigeonhole Problems

1.3 Using ANL-DP

Propositional input to ANL-DP is a sequence of clauses. (See 2.2 for input to the first-order model-searching program.) Literals are nonzero integers (negative integers represent negative literals), and each clause is terminated with 0. (So the entire input is just a sequence of integers.) The input is taken from `stdin` (the standard input).

ANL-DP accepts the following command-line options.

`-s`. Perform subsumption. (Subsumption is always performed during unit preprocessing.)

`-p`. Print models as they are found.

`-m` n. Stop when the n-th model is found.

`-t` n. Stop after n seconds.

`-k` n. Allocate at most n K bytes for storage of clauses.

`-x` n. Quasigroup experiment n. See Sec 2.5.

`-B` *file*. Back up assignments to a file.

`-b` n. Back up assignments every n seconds.

`-R` *file*. Restore assignments from a file. The file typically contains just the last line of a backup file. Other input, in particular the clauses, must be given exactly as in the original search.

`-n` n. This option is used for first-order model searches. The parameter n specifies the domain size and its presence tells the program to read first-order flattened relational input clauses instead of propositional clauses.

2 The First-Order Model-Searching Program

The first practical program for searching for small models of first-order statements was FINDER [6]. Another model-searching program is MGTP [7], which uses a somewhat different approach. The third class of programs, including LDPP [8], SATO [8], and the one described here, are based on Davis-Putnam procedures. None of these programs is clearly better than the others, and each has answered open questions about quasigroups (see Sec. 2.5).

The Davis-Putnam approach is quite elegant, because the computational engine—the Davis-Putnam code—is in no way tailored to first-order model searching. First-order clauses and a domain size n are input, then ground instances (over the domain) of the first-order clauses are generated and given to the Davis-Putnam code. Any propositional models that are found can be easily translated to first-order models (e.g., an $n \times n$ table for a binary function).

The steps, which are summarized in Figure 1, are as follows.

$$\begin{tikzcd}
\text{FO formula} \arrow[r, "(1)"] & \text{FO clauses} \arrow[r, "(2)"] \arrow[d, dotted] & \text{FO flat clauses} \arrow[r, "(3)"] & \text{Prop. clauses} \arrow[d, "(4)"] \\
& \text{FO models} & & \text{Prop. models} \arrow[ll, "(5)"']
\end{tikzcd}$$

Figure 1: Searching for First-Order Models

(1) Take an arbitrary first-order formula (possibly involving equality) and produce a set of clauses. OTTER's clausification code is sufficient for this.

(2) Take a set of first-order clauses and produce a set of flattened, relational clauses that contain no constants or function symbols—all arguments of the literals are variables. The steps are:

   a. For each n-ary function symbol (including constants), an $n+1$-ary predice symbol is introduced. For the examples, function symbols are lower case letters, and new predicate symbols are the corresponding upper case letters.

   b. To flatten the clauses, the following kind of equality transformation is applied to nonvariable terms (excepting arguments of positive equalities): $P[t]$ is rewritten to $t \neq x \mid P[x]$.

   c. A clause containing a positive equality $\alpha = \beta$, where both arguments are nonvariable, is made into two clauses: $L \mid \alpha = \beta$ becomes $L \mid \alpha \neq x \mid \beta = x$ and $L \mid \beta \neq x \mid \alpha = x$.

   d. Each functional literal, say $f(x, y) = z$, is rewritten into its relational form, say $F(x, y, z)$. (The resulting clauses may contain ordinary equality literals as well.)

   For example, the equality $f(g(x), x) = e$ produces the two clauses

$\neg G(x,y) \mid \neg E(z) \mid F(y,x,z)$
$\neg G(x,y) \mid E(z) \mid \neg F(y,x,z)$

(3) Take a set of flattened relational clauses and a domain size, and generate a set of propositional clauses. For each relational clause, the set of instances over the domain is constructed. (With domain size n, a clause with m variables produces n^m instances.) Each atom is encoded into a unique integer that becomes the propositional variable. Also, we must assert that the $(n+1)$-ary predicates introduced above represent total functions, so for each, we assert two sets of propositional clauses. For example, for ternary relation F, we must say that the last argument is a function of the others:

$\neg F(x,y,z_1) \mid \neg F(x,y,z_2)$, for $z_1 < z_2$ (well-defined)

and that the function is total and its value always lies in the domain (elements of the domain are named $0, 1, \cdots, n-1$):

$F(x,y,0) \mid F(x,y,1) \mid \cdots \mid F(x,y,n-1)$ (closed and total).

Nothing special needs to be done for ordinary predicate symbols.

(4) The Davis-Putnam procedure searches for models of the propositional clauses.

(5) For each propositional model, we generate the corresponding first-order model. The clauses given in (3) above ensure that from the propositional model, we can build a function for each function symbol (including constants).

2.1 Additional Constraints

For various reasons, the most important being to reduce the number of isomorphic models that are found, the user can specify part of the model by supplying ground clauses over the domain. For example, if a noncommutative group is being sought, with constants a and b as noncommuting elements, the user can assign 0 to the identity, 1 to a, and 2 to b. In this case, nothing is lost by making them distinct.

Symbols can be given the following properties.

`quasigroup`. This can be applied to ternary relations that represent binary fuctions. The multiplication table of a quasigroup has one of each element in each row and each column.

`bijection`. This can be applied to binary relations that represent unary fuctions.

`equality`. This can be applied to binary relations. It is just equality of domain elements.

`order`. This can be applied to binary relations. This is just the less-than relation on the domain elements.

`holey`. This can be applied to ternary relations that represent binary fuctions. See Sec. 2.5.2.

`hole`. This can be applied to binary relations. See Sec. 2.5.2.

2.2 Using the Program to Search for First-Order Models

The first-order searcher is part of ANL-DP, and it is invoked as described in Sec. 1.3. The command-line option "`-n` n" specifies the domain size and indicates that the input will be given as first-order flat clauses. Here is an example input specifying a noncommutative group.

```
function F 3 quasigroup
function E 1 -----
function G 2 bijection
function A 1 -----
function B 1 -----
end_of_symbols

-E v0   F v0 v1 v1 .
-E v0   -G v1 v2   F v2 v1 v0 .
E v0   -G v1 v2   -F v2 v1 v0 .
-F v0 v1 v2   -F v3 v2 v4   -F v3 v0 v5   F v5 v1 v4 .
-F v0 v1 v2   F v3 v2 v4   -F v3 v0 v5   -F v5 v1 v4 .
-F v0 v1 v2   -B v0   -A v1   -F v1 v0 v2 .
end_of_clauses

E 0
A 1
B 2
end_of_assignments
```

Symbol Declarations. In the first section of the input, each symbol is declared with four strings: type (`function` or `relation`), symbol, arity (n+1 for functions), and properties (`equality`, `order`, `quasigroup`, `bijection`, or `-----`).

Clauses. Flat relational clauses appear in the second section. Variables can be any strings. *Whitespace is required before the periods that terminates clauses.*

Assignments. Ground units (without periods) can appear in the third section.

2.3 Using Otter to Generate the Flat Clauses

Otter 3.0.2 and later versions can take ordinary formulas or clauses and produce the flat relational clauses for input to ANL-DP. Here is an Otter input file for a noncommutative group that will produce something like the file in Sec. 2.2.

```
set(dp_transform).

list(usable).
f(e,x) = x.
f(g(x),x) = e.
f(f(x,y),z) = f(x,f(y,z)).
f(a,b) != f(b,a).
```

```
end_of_list.

list(passive).
properties(f(_,_), quasigroup).
properties(g(_), bijection).
assign(e, 0).
assign(a, 1).
assign(b, 2).
end_of_list.
```

The command `set(dp_transform)` tells OTTER to generate input for an ANL-DP search and then exit.

The output of OTTER contains extraneous text, so it must be passed though a filter before ANL-DP can receive it. See the example files and scripts in the distribution directories.

2.4 The Order Relation

The ordered semigroup example in the FINDER 3.0 manual [6, Sec. 4.1.5] motivated me to have ANL-DP recognize the less-than relation on domain elements. The input (in OTTER form) for the ordered semigroup problems is as follows.

```
set(dp_transform).
list(usable).
f(f(x,y),z) = f(x,f(y,z)).
-(f(x,y) < f(x,z)) | y < z.
-(f(y,x) < f(z,x)) | y < z.
end_of_list.
```

Table 2 compares the results of FINDER and ANL-DP, both run on SPARC 2 computers, on the ordered semigroup problems. FINDER's search algorithm was developed with this type of problem in mind; ANL-DP simply adds the n^2 unit clauses for the less-than relation. I believe this explains most of the disparity of the times.

Order	Models	FINDER	ANL-DP
3	44	0.1	0.1
4	386	0.6	2.6
5	3852	9.2	58.0

Table 2: Ordered Semigroup Problems – FINDER vs. ANL-DP

2.5 Application to Quasigroup Problems

In the multiplication table of an order-n quasigroup, each row and each column is a permutation of the n elements. For these problems, we are interested only in idempotent (i.e., $xx = x$) models. Additional constraints are given for the seven problems listed in Table 3. (Notes: (1) For QG1 and QG2, the disjunction to the right of the implication is ordinarily a conjunction; the forms are equivalent for quasigroups, and models are found more easily with disjunction. (2) The second and third equalities for QG5 and the second equality for QG7 are dependent.) See [2] and [7] for details on the quasigroup problems.

Name	Constraints
QG1	$xy = u \wedge zw = u \wedge vy = x \wedge vw = z \rightarrow x = z \vee y = w$
QG2	$xy = u \wedge zw = u \wedge vx = y \wedge vz = w \rightarrow x = z \vee y = w$
QG3	$(xy)(yx) = x$
QG4	$(xy)(yx) = y$
QG5	$((xy)x)x = y \wedge x((yx)x) = y \wedge (x(yx))x = y$
QG6	$(xy)y = x(xy)$
QG7	$((xy)x)y = x \wedge ((xy)y)(xy) = x$

Table 3: The Quasigroup Problems

We also used the following cycle constraint on the last column to eliminate some isomorpohic models [7]:

$\neg f(x, n, z)$, for $z < x - 1$.

The constraint requires that cycles in the last column be made up of contiguous elements. This constraint is specified to ANL-DP with the command-line option "`-x1`"; *the quasigroup operation must be* `f` *(lower case) for this to work.*

Table 4 gives summaries of the performance of ANL-DP (C, list structure), SATO-2 (C, trie structure), and LDPP′ (Lisp, list structure) on some cases of the quasigroup problems. The SATO and LDPP figures are taken from [8]. All runs were made on a SPARC 2 or similar computer. All programs used the cycle constraint and similar selection functions for splitting. I believe that differences in the number of branches are due mostly to the order of clauses and literals. Search time is given in seconds.

Table 5 lists some additional statistics for ANL-DP on the quasigroup problems. "Generated" and "Searched" are the number of propositional clauses generated and the number remaining after subsumption and the initial unit propagation. "Create" is the time (seconds) used to construct the propositional clauses.

2.5.1 Cyclically-Generated Quasigroups

The command-line option `-x2` constrains models of quasigroup `f` to have the property $f(x+1, y+1) = f(x, y) + 1$, where addition is (mod n); that is, all the diagonals count up (mod *domain-size*).

The command-line option `-x`i, where $11 \leq i \leq 19$, constrains models of quasigroup `f` in the following way. Consider the square of size $x = i - 10$ in the lower right corner and the remaining square of size $m = n - x$ in the upper left corner. The diagonals of the upper left square count up (mod m), except for diagonals that consist of the same element in $m, \cdots, n-1$. Also, the first m elements of the last x rows and columns count up (mod m). For example (see [2, Example 8.1] with the input (note that the only upper left corner is idempotent)

```
set(dp_transform).
```

Problem	Models	ANL-DP Branches	ANL-DP Search	SATO-2 Branches	SATO-2 Search	LDPP′ Branches	LDPP′ Search
QG1.7:	8	388	2.05	376	1	389	26
.8:	16	100731	852.81	102610	379	101129	3463
QG2.7:	14	361	2.23	340	1	205	8
.8:	2	77158	810.75	80245	341	33835	1358
QG3.8:	18	1017	2.82	1072	3	573	5
.9:	-	39461	155.12	48545	157	24763	208
QG4.8:	-	891	2.40	925	2	602	4
.9:	178	52939	209.76	52826	168	27479	228
QG5.9:	-	14	.22	19	.2	15	.4
.10:	-	37	.52	62	.5	38	.9
.11:	5	112	2.16	111	2	125	5
.12:	-	369	6.61	369	7	369	15
.13:	-	9588	242.54	10764	224	12686	639
QG6.9:	4	17	.25	24	.2	18	.4
.10:	-	58	.54	150	.7	59	.8
.11:	-	537	5.36	519	6	539	11
.12:	-	7306	95.41	5728	92	7288	177
QG7.9:	4	7	.19	7	.2	8	.3
.10:	-	39	.38	54	.4	40	.7
.11:	-	291	2.98	254	3	294	6
.12:	-	1578	17.87	1281	22	1592	38
.13:	64	33946	493.67	27988	592	34726	1050

Table 4: Quasigroup Problems – Comparison

```
list(usable).
% (3,1,2)-COLS
f(x,y)!=u | f(z,w)!=u | f(v,x)!=y | f(v,z)!= w | x=z | y=w.
end_of_list.

list(passive).
properties(f(_,_), quasigroup).
assign(f(0,0),0).
end_of_list.
```

and the options "`-n 10 -x13 -p`", we get

```
Model #1 at 333.47 seconds (SPARC 10):

f    | 0 1 2 3 4 5 6  7 8 9
--------------------------
   0 | 0 4 1 7 9 2 8  3 6 5
   1 | 8 1 5 2 7 9 3  4 0 6
   2 | 4 8 2 6 3 7 9  5 1 0
   3 | 9 5 8 3 0 4 7  6 2 1
   4 | 7 9 6 8 4 1 5  0 3 2
   5 | 6 7 9 0 8 5 2  1 4 3
   6 | 3 0 7 9 1 8 6  2 5 4
     |
```

Problem	Models	Branches	Generated	Searched	Memory	Create	Search
QG1.7	8	388	120954	8952	886 K	4.79	2.05
.8	16	100731	267805	28877	2061 K	10.85	852.81
QG2.7	14	361	120954	9830	886 K	4.72	2.23
.8	2	77158	267805	30902	2061 K	10.88	810.75
QG3.8	18	1017	9757	3830	303 K	0.26	2.82
.9	-	39461	15670	6966	601 K	0.39	155.12
QG4.8	-	891	9757	3830	303 K	0.25	2.40
.9	178	52939	15670	6966	601 K	0.37	209.76
QG5.9	-	14	28792	9694	894 K	0.85	0.22
.10	-	37	43946	17274	1193 K	1.39	0.52
.11	5	112	64428	28488	1786 K	2.05	2.16
.12	-	369	91363	44302	2674 K	2.93	6.61
.13	-	9588	125984	65790	3562 K	4.02	242.54
QG6.9	4	17	22231	7653	601 K	0.66	0.25
.10	-	58	33946	13579	900 K	1.02	0.54
.11	-	537	49787	22332	1493 K	1.54	5.36
.12	-	7306	70627	34662	2088 K	2.24	95.41
QG7.9	4	7	22231	5838	601 K	0.61	0.19
.10	-	39	33946	11038	900 K	1.04	0.38
.11	-	291	49787	18944	1493 K	1.48	2.98
.12	-	1578	70627	30309	2088 K	2.14	17.87
.13	64	33946	97423	45967	2683 K	3.14	493.67

Table 5: Quasigroup Problems – ANL-DP Full Statistics

```
 7 | 1 2 3 4 5 6 0  7 8 9
 8 | 2 3 4 5 6 0 1  8 9 7
 9 | 5 6 0 1 2 3 4  9 7 8
```

The `-x`*i* option can also be used when searching for quasigroups with holes.

2.5.2 Quasigroups with Holes

We simply list an example. The input (compare with above input)

```
set(dp_transform).

list(usable).
same_hole(x,x) | f(x,x) = x.
% (3,1,2)-COLS
f(x,y)!=u | f(z,w)!=u | f(v,x)!=y | f(v,z)!= w | x=z | y=w.
end_of_list.

list(passive).
properties(f(_,_), quasigroup_holey).
properties(same_hole(_,_), hole).
% The program makes same_hole symmetric and transitive.
assign(same_hole(7,8), T). assign(same_hole(8,9), T).
end_of_list.
```

with the command-line options "-n10 -x13 -p" produces

```
Model #1 at 50.07 seconds (SPARC 2):

f    | 0 1 2 3 4 5 6 7 8 9
--------------------------
   0 | 0 6 7 5 8 9 3 1 2 4
   1 | 4 1 0 7 6 8 9 2 3 5
   2 | 9 5 2 1 7 0 8 3 4 6
   3 | 8 9 6 3 2 7 1 4 5 0
   4 | 2 8 9 0 4 3 7 5 6 1
   5 | 7 3 8 9 1 5 4 6 0 2
   6 | 5 7 4 8 9 2 6 0 1 3
   7 | 3 4 5 6 0 1 2 - - -
   8 | 6 0 1 2 3 4 5 - - -
   9 | 1 2 3 4 5 6 0 - - -
```

2.5.3 Open Quasigroup Questions Answered

Orthogonal Mendelsohn Triple Systems (OMTS). Corollary 5.2 of [3] states

> The necessary condition for the existence of a pair of OMTS(v), that is, $v \equiv 0$ or 1 (mod 3), is also sufficient except for v=3,6 and possibly excepting $v \in \{9, 10, 12, 18\}$.

See [3] for definitions. The input

```
set(dp_transform).

list(usable).
f(x,x) = x.
h(x,x) = x.
f(x,f(y,x))=y.
h(x,h(y,x))=y.
f(x,y)!=u | f(z,w)!=u | h(x,y)!=v | h(z,w)!=v | x=z | y=w.
end_of_list.

list(passive).
properties(f(_,_), quasigroup).
properties(h(_,_), quasigroup).
end_of_list.
```

with the options "-n9 -x1 -p" produces the following quasigroups, which correspond to a pair of orthogonal Mendelsohn triple systems of order 9.

```
Model #1 at 54.58 seconds (SPARC 2):

f    | 0 1 2 3 4 5 6 7 8        h    | 0 1 2 3 4 5 6 7 8
------------------------        ------------------------
   0 | 0 8 5 2 7 4 3 6 1           0 | 0 2 6 4 3 1 8 5 7
   1 | 8 1 7 6 3 2 5 4 0           1 | 5 1 0 8 2 3 7 6 4
   2 | 3 5 2 8 6 0 4 1 7           2 | 1 4 2 6 7 8 0 3 5
   3 | 6 4 0 3 8 7 1 5 2           3 | 4 5 7 3 0 6 2 8 1
```

```
4 | 5 7 6 1 4 8 2 0 3          4 | 3 8 1 0 4 7 5 2 6
5 | 2 6 1 7 0 5 8 3 4          5 | 7 0 8 1 6 5 3 4 2
6 | 7 3 4 0 2 1 6 8 5          6 | 2 7 3 5 8 4 6 1 0
7 | 4 2 8 5 1 3 0 7 6          7 | 8 6 4 2 5 0 1 7 3
8 | 1 0 3 4 5 6 7 2 8          8 | 6 3 5 7 1 2 4 0 8
```

An analogous search for OMTS(10) ran for several days without finding a model.

QG3(2^8). A quasigroup of type h^n has order $h * n$ and n holes of size h. Frank Bennett [1] posed the question of the existence of QG3(2^8). (Mark Stickel had already answered positively the question of the existence of QG3(2^6).) The ANL-DP input

```
relation = 2 equality
relation same_hole 2 hole
function f 3 quasigroup_holey
end_of_symbols

f v0 v0 v0   same_hole v0 v0 .
-f v0 v1 v2   -f v1 v0 v3   f v3 v2 v1 .
-f v0 v1 v2   -f v0 v2 v1   = v0 v1 .
-f v0 v1 v2   -f v2 v1 v0   = v0 v1 .
end_of_clauses

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

with the options "`-n16 -p`" produces the following holey quasigroup.

```
Model #1 at 76086.21 seconds (i468 DX2/66):

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

```
   15 | 12 11  5  7  8 13  2  4  0 10  1  9  6  3  -  -
```

(In case the reader is wondering why the holes are irregular in the lower right corner, the reason is that preliminary runs on QG3(2^6) ran faster with a similar hole configuration than with a regular configuration.)

QG7(17,5). Frank Bennett posed [1] the question of whether the the quasigroup identity (QG7a) $x(yx) = (yx)y$ implies either $(xy)x = x(yx)$ or $xy(yx) = y$. He suggested looking at models of order 17 with a hole of size 5, if they exist, as possible counterexamples. The identity (QG7b) $((xy)x)y = x$, which is conjugate-equivalent [2] to (QG7a), is much easier to work with, so we put ANL-DP to work with the input

```
relation same_hole 2 hole
function f 3 quasigroup_holey
end_of_symbols

f v0 v0 v0  same_hole v0 v0 .
-f v0 v1 v2   -f v2 v0 v3   f v3 v1 v0 .
end_of_clauses

same_hole 12 13
same_hole 13 14
same_hole 14 15
same_hole 15 16
end_of_assignments
```

and the options "`-n17 -x1 -p`", which produced

```
Model #1 at 172914.77 seconds (SPARC 2):

f    |  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
-----------------------------------------------------------
   0 |  0  2  1 16 13 11 12  8  5 15 14  7  4  6  9 10  3
   1 | 16  1  3  2 10 13  9 12 15  4  6 14  5  7  8 11  0
   2 |  3 16  2  0 15  9 14 10 12  7  5 13 11  8  4  6  1
   3 |  1  0 16  3  8 15 11 14  6 12 13  4 10  9  5  7  2
   4 |  8 13 10 15  4  6  5 16  2 14  0 12  9  3 11  1  7
   5 | 13  9 15 11 16  5  7  6 14  3 12  1  8  2 10  0  4
   6 | 11 12  9 14  7 16  6  4 13  0 15  2  3 10  1  8  5
   7 | 12 10 14  8  5  4 16  7  1 13  3 15  2 11  0  9  6
   8 | 15  6  5 12 14  1  2 13  8 10  9 16  0  4  7  3 11
   9 |  7 15 12  4  0 14 13  3 16  9 11 10  1  5  6  2  8
  10 |  5 14 13  6 12  3  0 15 11 16 10  8  7  1  2  4  9
  11 | 14  4  7 13  2 12 15  1  9  8 16 11  6  0  3  5 10
  12 | 10 11  4  5  3  2  8  9  7  6  1  0  -  -  -  -  -
  13 |  9  8  6  7 11 10  3  2  0  1  4  5  -  -  -  -  -
  14 |  4  5  8  9  1  0 10 11  3  2  7  6  -  -  -  -  -
  15 |  6  7 11 10  9  8  1  0  4  5  2  3  -  -  -  -  -
  16 |  2  3  0  1  6  7  4  5 10 11  8  9  -  -  -  -  -
```

The conjugate-equivalent quasigroup corresponding to (QG7a) was then generated and found to falsify the two identities in question, giving a counterexample to the problem.

References

[1] F. E. Bennett. Correspondence by electronic mail, 1994.

[2] F. E. Bennett and L. Zhu. Conjugate-orthoganal latin squares and related structures. In J. H. Dinitz and D. R. Stinson, editors, *Contamporary Design Theory: A Collection of Surveys*, pages 41–96. John Wiley & Sons, 1992.

[3] F. E. Bennett and L. Zhu. Self-orthogonal Mendelsohn triple systems. Preprint, 1994.

[4] M. Fujita, J. Slaney, and F. E. Bennett. Automatic generation of some results in finite algebra. In *International Joint Conference on Artificial Intelligence*, 1993.

[5] W. McCune. OTTER 3.0 Reference Manual and Guide. Tech. Report ANL-94/6, Argonne National Laboratory, Argonne, Ill., 1994.

[6] J. Slaney. Finder version 3.0 notes and guide. Tech. report, Centre for Information Science Research, Australian National University, 1993.

[7] J. Slaney, M. Fujita, and M. Stickel. Automated reasoning and exhaustive search: Quasigroup existence problems. *Computers and Mathematics with Applications*, 1994. To appear.

[8] H. Zhang and M. Stickel. Implementing the Davis-Putnam algorithm by tries. Preprint, 1994.