THE CRIPPLED QUEEN PLACEMENT PROBLEM

Robert WAGNER and Robert GEIST*

Department of Computer Science, Duke University, Durham, NC 27706, U.S.A.

Communicated by R. Sethi
Received October 1983
Revised January 1984

Abstract. We describe the outcome of various combinations of choices made by individuals in the solution of a non-trivial combinatorial problem on a computer. The programs which result are analyzed with respect to execution speed, design time, and difficulty in debugging. The solutions obtained vary dramatically as a result of choices made in the overall design of the solution. Choices made at lower levels in the top-down tree of design choices seem to have less effect on the parameters analyzed. A tradeoff between mathematical effort in algorithm design, and program speed is evident, since some solutions required solution-time which grows exponentially with the case size, while another solution presented here gives a closed-form expression for the required answers for all large cases.

Introduction

This paper is an attempt to document the major choices made in designing a computer program to solve a non-trivial problem, a variation of the classical eight queens problem [1, 3], originally posed by Franz Nauck in 1850. Like the eight queens problem, it is representative of a large class of combinatorial problems and is mathematically interesting in its own right. We hope that it is also representative of problems encountered by programmers in practice.

In presenting solutions, we attempt to exhibit tradeoffs between two principal categories of variables:

(1) The effort expended in mathematical analysis of the problem, before code (or even coding specification) is developed, and
(2) The 'quality' of the resulting program product, measured in terms of
   (a) execution speed,
   (b) textual compactness and clarity,
   (c) difficulty in debugging (more generally, difficulty in obtaining acceptably high confidence in program correctness).

We observe that choices made at high levels of abstraction have much larger impact on quality and involve a much higher level of analytic effort than choices.

* Current address: Department of Computer Science, Clemson University, Clemson, SC 29631, U.S.A.
made to solve 'subproblems'. In particular, we observe large improvements in execution speed as a result of additional analytic effort. This is accompanied by little or no change in measures of textual acceptability, and/or debugging difficulty.

In contrast, one subproblem, generated to help implement one solution technique, required large inputs of analytic effort, but generated a large percentage of the program text and the debugging difficulties observed. Moreover, solving this sub-problem probably contributed a comparatively small speed improvement.

We conclude by suggesting that, in the hierarchical development of a program, whenever possible, one should attempt to judge execution speed and conceptual difficulty of implementing a (sub) program, before proceeding to the next stage of hierarchical refinement. At the moment, some analytic tools for estimating worst-case run-time are available; however, no simple, trustworthy methods for average run-time estimation are known, primarily because the probability of occurrence of any identifiable subclass of problem instances is unknown. Also, we know of no quantitative method of estimating, in advance, measures of program complexity, understandability, and debuggability. Thus, our planning is haphazard and subject to error in these areas. By studying the program development process, we hope to shed some light on the choices made during program development that control the performance, complexity, and size of implementation of the resulting product.

The remainder of the paper is organized as follows. The problem and its straightforward solution are presented in Section 1. The straightforward solution is readily seen to be unacceptable, and so a necessary improvement is also presented there. In Section 2, a deeper analysis of the problem is combined with the technique of Dynamic Programming [2, 4] to yield first a modest success and then a substantial improvement thereon. In Section 3, geometric reduction and mathematical induction are applied to obtain a closed form solution for most problem instances. Comparisons and conclusions are presented in Sections 4 and 5 respectively.

1. The problem

The following problem was presented to a class of graduate students in Computer Science. It was chosen by one of the authors (Wagner) as a natural generalization of the familiar 8-queens problem [1]. The problem was chosen to be 'non-trivial', in that it was hoped that many programs using distinctly different strategies could be invented for its solution.

"A Crippled Queen (CQ) is a chess queen that can move at most 2 squares at a time in any queen-attack direction (vertical, horizontal, or diagonal). Find the number $C_{88}$ of such CQ's that can be placed on an 8 x 8 chess board so that no two attack one another. Find also $N_{88}$, the number of different ways $C_{88}$ CQ's can be so placed. If possible, generalize the program to compute $C_{8n}$ and $N_{8n}$ or $C_{mn}$ and $N_{mn}$, defined as above, but for an 8 x n or m x n chessboard. Your program's run time will be one criterion for determining its grade."
1.1. Its straightforward solution

Most of the class wrote straightforward generalizations of the simplest 8-queens solution. In this backtracking approach, a recursive routine searches over a linear ordering of the \( m \times n \) squares for the next unattacked square, marks that square occupied, marks appropriate adjacent squares attacked, and calls itself to proceed with the search from that point. When the board is full, the routine recursively removes the last-placed CQ and seeks the next possible location for it in the linear ordering. We present this and all other programs in a 'pidgin' version of C [5]:

**Solution 1.**

```c
global integer BOARD[M * N]; /* indicates which squares are currently occupied, and/or attached. */

main ( ) {
Cmn = 0; /* maximum number of CQ's placed */
Nmn = 1; /* number of solutions with Cmn CQ's */
place(1);
}

place(sq)
integer sq;
local integer s;
s = sq;
while(s <= m * n){
    if (BOARD[s] is unattacked by previously placed CQ's)
        {Mark BOARD[s] occupied;
         place(nextsq[s]);
         Mark BOARD[s] unoccupied;}
    s = nextsq[s];
}
/* Board is full */
C = # squares marked occupied;
if (C > Cmn) {Cmn = C; Nmn = 1};
if (C == Cmn) Nmn = Nmn + 1;
return; /* from place() */
}
```

Here nextsq[s] gives the square numbered next after \( s \) in the linear ordering of the \( m \times n \) squares.

1.2. A necessary improvement

Unfortunately, while Solution 1 satisfies the originally-stated specifications, it is not acceptable: its execution for the 8 x 8 case uses from 40 to 240 minutes of PDP-11/70 time.
Experiments showed that execution time could be dramatically improved by refusing to extend a partial solution that placed, say, \( k \) CQ's in the first \( x \) columns of the board if \( k + C_{m(n-x)} < C_{mn} \), where, as before, \( C_{mj} \) denotes the maximum number of CQ's placeable on an \( m \) row by \( j \) column board.

Execution times for solutions incorporating this 'branch-and-bound' strategy ranged from 4.8 to 35.7 seconds on the PDP-11/70, for the \( 8 \times 8 \) case. A typical version, incorporating the suggested modification is:

**Solution 2.**

```c
global integer BOARD[m*n], Cmn, Nmn, Cm[n+1];
global integer j; /* columns in current board */
global integer k; /* CQ's placed in first x cols of current board */

main( ) {
  for (j = 1; j <= n; j = j + 1) {
    k = 0;
    Cmn = 0;
    Nmn = 1;
    place(1):
    Cm[j] = Cmn;
  } /* There are 'Nmn' solutions which place 'Cmn' CQ's on the 'm' by 'n' chessboard */

  place(sq){
    integer sq;
    local integer s,
    s = sq;
    while( s <= m * j ) {
      if (k + Cm[ j - column[s]] < Cmn) break;
      if (BOARD[s] is unattacked by previously placed CQ's)
        {Mark BOARD[s] occupied;
         k = k + 1;
         place(nextsq[s]);
         k = k - 1;
         Mark BOARD[s] unoccupied:}
      s = nextsq[s];
    } /* m by j board is full */
    if (k > Cmn) {Cmn = k; Nmn = 1:}
    if (k = = Cmn) Nmn = Nmn + 1;
    return;
  }
```

Here, column[s] gives the column number (from 1 to n) of the column in which square s occurs. Both nextsq and column are arrays, which can be initialized before the main computation begins.

The fastest versions of this strategy caused the “Mark board[s] occupied” step to also mark all squares t such that t > s, and a CQ on square s attacks square t as ‘attacked (again)’, usually by incrementing an ‘attack count’ associated with each such square. To simplify the edge-of-board problems, the array board was often augmented to an \((m+4)\) by \((n+4)\) array, allowing the ‘real’ board to be bordered by two rows and two columns of ‘off-board’ squares on each side. Slower versions of the program failed to include the loop to calculate \(C_m[j]\), but instead relied on

(a) a formula, such as \(C_m[j] \leq 3j\) (for the \(m = 8\) case)—no more than 3 CQ’s can be placed in one column without mutual attacks; or

(b) re-running the program by hand for several successive values of \(n\), and filling in the \(C_m[]\) table using the results of those runs manually.

A plot of the time needed to solve the \(8 \times n\) case showed that all programs using this strategy ran in exponential time as a function of \(n\). In fact, under a 10-minute CPU time limit, the case \(8 \times 16\) proved to be out of reach for even the fastest program.

2. A deeper analysis?

Is it possible to further improve the speed-of-solution of an algorithm for the CQ problem?

Readers interested in problem-solving techniques are urged at this point to answer this question for themselves, before reading further sections. They may assume that solutions for large values of \(m\), or of \(n\), or of \(m \times n\) are needed for some reason, and that run time for these generalized problems are to be limited to no more than 10 minutes of CPU time on a PDP-11/70.

2.1. A deeper analysis

Dynamic Programming [2, 4] can sometimes be applied to combinatorial problems to yield significant gains in speed-of-solution. Unfortunately, it is often difficult to see how to apply it, even when the technique is familiar to the analyst/programmer. In the present case, considerable thought was expended on several blind alleys before achieving some modest success:

**Blind alley 1.** Consider solving the CQ problem on an \((n+1)\) by \((n+1)\) board, given some (constrained) solution-information for the \(i \times i\) boards, \(i \leq n\).

This approach proved unpromising, since it appeared that

(a) there were a large number of ways of bordering an \(n \times n\) board with patterns of non-attacking CQ’s in a border row-and-column;
(b) the constraints imposed by each such border pattern amounted to deleting certain ragged patterns of edge squares from the $n \times n$ board; initial analysis seemed to show that some $4^n$ different 'ragged boards' of maximum dimension $n \times n$ might have to be considered.

Blind alley 2. Try increasing the size of a board by one square, instead of bordering an $n \times n$ board with an entire row-and-column of CQ's. This approach seemed worse than $\neq 1$, and was quickly (perhaps too quickly?) abandoned.

2.2. Modest success

Try solving the $m \times n$ board in terms of constrained solutions to the $m \times (n - 1)$ board. The most important conceptual milestones in the development of this solution were:

1. an experimental determination (Anselmo Lastra), that there were only 28 different non-attacking CQ placement patterns on the $8 \times 1$ board;

2. a realization that the 'constraints' needed on the $m \times (n - 1)$ board could be described compactly by fixing the CQ patterns present in the last two columns of the $m \times (n - 1)$ board. The 'legality' of a specific pattern occurring in the $n$th column can then be determined.

The following reformulation of the CQ problem resulted (we number columns from right to left for this formulation): Fix $m$ and let $C_m(S, T, i)$ be the maximum number of CQ's placeable on an $m$ row by $i$ column board whose last and next to last columns hold CQ patterns 'S' and 'T' respectively, where $i \geq 2$. If we let $X \parallel Y$ ($X \parallel Y$) denote the binary relation on pattern numbers which is true if and only if pattern number $X$ can be placed 2 columns (1 column) away from pattern number $Y$ without CQ attacks resulting, then for $i > 2$, we have

$$C_m(R, S, i) = \max_T \{|R| + C_m(S, T, i - 1) : S \parallel T \text{ and } R \parallel T\},$$

if $R | S$, and 0 otherwise, where $|R|$ = number of CQ's in pattern number $R$. Clearly $C_m(R, S, 2) = |R| + |S|$, if $R | S$, and 0 otherwise, and $\max_{(R,S)} \{C_m(R, S, n) : R | S\}$ is the maximum number of CQ's placeable on the $m \times n$ board.

Furthermore, if we define $N_m(R, S, i)$ = number of solutions to the CQ placement problem on the $m \times i$ board containing $C_m(R, S, i)$ CQ's, where the last two columns of the board hold patterns numbered $R$ and $S$, respectively, then $N_m(R, S, i)$ can then be computed as follows:

$$N_m(R, S, 2) = \begin{cases} 1 & \text{if } R | S, \\ 0 & \text{otherwise}, \end{cases}$$

and for $i > 2$,

$$N_m(R, S, i) = \left\{ \begin{array}{ll} \sum_T \{N_m(S, T, i - 1) : S \parallel T, R \parallel T \} & \text{if } R | S, \\ \text{and } |R| + C_m(S, T, i - 1) = C_m(R, S, i) \} & \text{otherwise}. \end{array} \right.$$
The reader should find it easy to verify that, if an \( m \times i \) board \((i > 2)\) holds a maximal number of non-attacking CQ's, and its last two columns hold patterns \( R \) and \( S \) respectively, then some pattern \( T \) occupies column \( i - 2 \) such that: \( R|S \) and \( S|T \) and \( R\|T \), and columns 1 to \( i - 1 \) contain \( \text{Cm}(S, T, i - 1) \) CQ’s. (For they could never contain more than this number of CQ’s, since \( \text{Cm}(S, T, i - 1) \) is the maximum number of CQ’s so placeable by definition: while if they contained fewer CQ's, the value of \( \text{Cm}(R, S, i) \) could be increased, contradicting its definition.)

We have thus established

**Solution 3.**

\[
\begin{align*}
&\text{for } (i = 2; i \leq n; i = i + 1)\{ \\
&\quad \text{for (all } (R, S) \text{ such that } (R|S) \} \\
&\quad \quad \text{use above relations to tabulate } \text{Cm}[R, S, i] \text{ and } \text{Nm}[R, S, i] \\
&\quad \}\ \\
&\text{Cmn} = 0; \\
&\text{Nm} = 0; \\
&\text{for (all } R, S \text{ such that } (R|S) \} \\
&\quad C = \text{Cm}[R, S, n]; \\
&\quad \text{if } (C > \text{Cmn}) \{ \text{Cmn} = C; \text{Nm} = 0; \} \\
&\quad \text{if } (C = = \text{Cmn}) \text{ Nmn} = \text{Nm} + \text{Nm}[R, S, n]; \\
&\}
\]

For fixed \( m \), let \( P_m \) be the set of legal patterns for the \( m \times 1 \) board, that is, the set of solutions to the problem of placing any number of CQ's on the \( m \times 1 \) board without mutual attacks. Then assuming that \( X\|Y \) and \( X\|Y \) have been pre-tabulated for all \( X, Y \in P_m \), solution 3 requires running time \( O(p_m^3 n) \), where \( p_m = |P_m| \). Since \( p_m \approx 28 \), this gives an \( O(28^3 n) \) run-time for the \( 8 \times n \) case of the CQ problem. In general, the run-time for this solution is linear in \( n \). We estimate that about 20 \( \mu \)sec would be used per iteration of the inner loop, giving a ‘slope’ of about \( 20 \times 22000 \times 10^{-6} = 0.44 \) seconds per column. In 10 minutes, or 600 seconds, the case \( 8 \times 1300 \) could be solved, using space for about \( 1300 \times 28 \times 28 = 1020000 \) integers. (In practice, since only the tables for \( i \) and \( i - 1 \) need be present when computing the \( \text{Cm}[ ] \) and \( \text{Nm}[ ] \) tables for \( i \), required storage is independent of \( n \).)

This scheme is somewhat limited, however, since \( p_m \approx 2^{m/3} \). This can easily be seen as follows. \( p_m \) gives the number of placements of CQ's possible in one column of \( m \) squares without mutual attacks. But CQ's placed 3 squares apart never attack. Therefore, CQ's may be placed on any subset of the squares 1, 4, 7, 10, \ldots 3 \times \lceil (m - 1)/3 \rceil + 1, assuming squares not in this collection are vacant. There are \( \lceil (m - 1)/3 \rceil + 1 \approx m/3 \) squares in this set, giving a subclass of solutions of size at least \( 2^{m/3} \).

This limitation suggests solving an interesting subproblem: Solution 3 requires indexing over the set \( W_m = \{(R, S, T): R|S \text{ and } S|T \text{ and } R\|T\} \). Can the total number of index operations required be reduced substantially below \( p_m^3 \)? Indeed.
In the Appendix we provide the details of a technique for minimizing the required indexings. The analysis there also shows that \( w_m = |W_m| \), the size of the set \( W_m \), is \( O(2.21^n) \), so that solution time for the \( m \) row by \( n \) column CQ placement problem becomes \( O(n \times 2.21^n) \). There, we show directly that \( w_8 = 950 \), and hence that the revised version of Solution 3 should run a factor of \( p_8^2/w_8 = 28^2/950 \approx 23 \) faster than the original.

For this revised version of Solution 3, the empirical evidence shown in Table 1 was gathered to estimate the proportionality constant in the case \( m = 8 \).

<table>
<thead>
<tr>
<th>Board size</th>
<th>CPU time</th>
<th># CQ's</th>
<th># Placements</th>
</tr>
</thead>
<tbody>
<tr>
<td>8( \times )8</td>
<td>0.6</td>
<td>13</td>
<td>40</td>
</tr>
<tr>
<td>8( \times )16</td>
<td>0.9</td>
<td>26</td>
<td>18</td>
</tr>
<tr>
<td>8( \times )32</td>
<td>1.4</td>
<td>52</td>
<td>2</td>
</tr>
<tr>
<td>8( \times )48</td>
<td>2.0</td>
<td>77</td>
<td>64</td>
</tr>
<tr>
<td>8( \times )64</td>
<td>2.5</td>
<td>103</td>
<td>4</td>
</tr>
<tr>
<td>8( \times )80</td>
<td>3.1</td>
<td>128</td>
<td>472</td>
</tr>
<tr>
<td>8( \times )96</td>
<td>3.6</td>
<td>154</td>
<td>50</td>
</tr>
</tbody>
</table>

The CPU times are in seconds on our PDP-11/70 UNIX system. All routines were coded in C [5], and compiled by release 7 of Bell Telephone Labs compiler for that language.

Thus, we require about 0.5 seconds per 16 columns, plus 0.4 seconds per run, bearing out our statement that run-time is linear in \( n \), and showing that the proportionality constant is quite small for \( m = 8 \).

### 3. A closed-form solution for large boards

In this section we present an alternative approach, based upon the idea of board-partitioning, which ultimately yields closed-form solutions for sufficiently large boards. The initial motivation for this approach came from a desire to give a completely machine-free proof that \( C_{88} = 13 \). Accordingly, we present that proof first.

#### 3.1. \( C_{88} \)

Consider an infinite board of CQ's which appear to be rather tightly packed (Fig. 1). Upon attempting to over-lay an \( 8 \times 8 \) square so as to capture a maximum number of Q's, one quickly determines that \( C_{88} \) is certainly at least 13 and probably at most 13. Further, movement of the \( 8 \times 8 \) square over the infinite plane tends to emphasize the existence of constraints on sub-boards, so that one is led to 'wish' for a decomposition of the \( 8 \times 8 \) board into \( n \) sub-boards which are 'mutually independent' in the sense that \( C_{88} = \sum_{k=1}^{n} C_k \) for the appropriate sub-board sizes \( S_k \). It took approximately an hour to find the \( S_k \)'s; here is the result:
The crippled queen placement problem

Theorem 1. $C_{88} \geq 13$.

Proof. See Fig. 2. □

Theorem 2. $C_{88} \leq 13$.

Proof. The proof will consist of 3 lemmas:

Lemma 1. $C_{33} \leq 2$.

Proof. Up to symmetry, there are 3 possible initial placements (see Fig. 3). Placement 1 clearly permits no additional placements, while 2 and 3 each permit one additional
placement in one of two positions. This completes Lemma 1 (actually, it shows $C_{33} = 2$.) □

Lemma 2. $C_{35} \leq 3$.

**Proof.** Suppose there are 4 CQ’s in the $3 \times 5$ board of Figure 4(a). Then one-half or the other (left or right side of dashed line) must contain at least 2 CQ’s; say the right half. Now let $n_i =$ number of CQ’s in column $i$, $i = 1, \ldots, 5$, and observe that $n_1 + n_2 + \left(\frac{1}{2}\right)n_3 \geq 2$, but by Lemma 1, $n_1 + n_2 + n_3 \leq 2$, and thus $n_3 = 0$. Up to symmetry we are then in the position of Fig. 4(b), and neither of the two possible placements of 2 Q’s in the left wing can coexist with the right wing. This completes Lemma 2.

![Fig. 4](image)

Lemma 3. $C_{55} \leq 5$.

**Proof.** Suppose there are 6 CQ’s in the $5 \times 5$ board of Fig. 5. Then one half or the other must contain at least 3; say the right half. Then we have $n_1 + n_2 + \frac{1}{2}n_3 \geq 3$, but by Lemma 2, $n_1 + n_2 + n_3 \leq 3$, and thus $n_3 = 0$. By symmetry, the middle row must also be empty, but then we are forced to place 6 CQ’s in 4 disjoint $2 \times 2$ boards, which is clearly impossible. This completes Lemma 3. □

![Fig. 5](image)

Now for the complete proof of Theorem 2, simply consider Fig. 6, and apply Lemmas 1, 2, 2, and 3; thus $C_{88} = 13$. □
3.2. $N_{ss}$

Note that in Section 3.1 we established the restrictions of Fig. 7(a). In placing the maximum number of CQ's, 13, we must then have $w + x + y + z = 13$ and hence $w = 2, x = 3, y = 3,$ and $z = 5$ (note that this proves $C_{35} = 3$ and $C_{55} = 5$). But now observe that we can put our $3 \times 3$ square in any of the 4 corners! The reader can easily check that this gives the additional placement restrictions shown in Fig. 7(b).

Thus an algorithm for quickly computing $N_{ss}$ presents itself: fix a choice for the 1 CQ that must appear in the central $2 \times 2$ block, and mark appropriate squares attacked. (We will multiply by 4 later, that is, we will rotate each solution obtained from fixing this central block through 90, 180, and 270 degrees, thus producing all solutions.) For the remaining 8 blocks, move, say counterclockwise, around the board alternating between the placement of an acceptable (CQ's not attacked) pattern for a $3 \times 3$ block (there are 8 such) and the placement of an acceptable pattern for a $3 \times 2$ block (there are 6 such). When further placement becomes impossible, change the last-placed pattern to the next on the list of patterns for that size block, and then proceed forward again.

The possible placements of a single CQ in a $3 \times 2$ block obviously can be conveniently indexed $1, 2, \ldots, 6$; the placements of 2 CQ's in a $3 \times 3$ block are also conveniently indexed by imposing the numbering system of Fig. 8, so that all eight placements are given by pairs of consecutive integers $(i, i + 1)$, $i = 1, 2, \ldots, 8$ (mod 8).

$$
\begin{array}{ccc}
3 & 8 & 5 \\
6 & 2 & \\
1 & 4 & 7
\end{array}
$$

Fig. 8.
Solution 4.

int sqr[4]: /* sqr[i] = pattern chosen for square 3 x 3 block i */
int rctngl[4]: /* rctngl[i] = pattern chosen for rectangular 3 x 2 block i */
int i, j, k;

main() {
  place central CQ; /* 1 of 4 positions */
  i = j = 1;
  repeat {
    square();
    rectangle();
  }
}

square() {
  while (pattern j is not acceptable for square block i and j <= 8) j = j + 1;
  if (j <= 8) {
    sqr[i] = j;
    place pattern j in square block i;
    k = 1;
  }
  else {
    i = i - 1;
    if (i <= 0) exit;
    remove pattern rctngl[i] from rectangular block i;
    k = rctngl[i] + 1;
  }
}

rectangle() {
  while (pattern k is not acceptable for rectangular block i and k <= 6) k = k + 1;
  if (k <= 6) {
    rctngl[i] = k;
    place pattern k in rectangular block i;
    j = 1;
    i = i + 1;
    if (i > 4) {print solution; j = 9;}
  }
  else {
    remove pattern sqr[i] from square block i;
    j = sqr[i] + 1;
  }
}
Here "place" and "remove" indicate routines to increment and decrement the counts of attacks on appropriate squares of the 8 x 8 board.

In 0.2 seconds of execution time this program produced 10 solutions, and thus we have \( N_{88} = 40 \). Again note that all 40 solutions can be seen by simply rotating each of the 10 produced through 90, 180, and 270 degrees.

3.3. \( C_{mn} \)

The program of Section 3.2 was tightly tuned to the 8 x 8 case and obviously does not directly extend; nevertheless, the central idea there, placement of CQ patterns in blocks, rather than individual CQ's in squares, certainly does.

The problem of selecting a block size was quickly resolved: in an initial effort to generalize to square boards, the obvious candidates, from the lessons of Section 3.1, were the 3 x 3 block (8 patterns) and the 5 x 5 block (74 patterns, found in a straightforward search of 0.7 seconds). Upon comparison of the number of leaves in the respective solution trees for the LCM board (15 x 15), \( 8^{25} \) versus \( 74^9 \), we chose the latter. A simple program for \( 5k \times 5k \) boards was then immediate:

```c
int blck[k^2]; /* blck[i] = pattern selected for ith block */
int i, j;

main ( ) {
  i = j = 1;
  repeat {
    while (pattern j is not acceptable for block i and j \leq 74) j = j + 1;
    if (j \leq 74) {
      blck[i] = j;
      place pattern j in block i;
      i = i + 1;
      j = 1;
      if (i > k^2) {print solution; j = 75;}
    } else {
      i = i - 1;
      if (i \leq 0) exit;
      remove pattern blck[i] from block i;
      j = blck[i] + 1;
    }
  }
}
```

Of course, there was no a priori guarantee that this program would produce any solutions to the \( 5k \times 5k \) board (\( k > 1 \)), for, although \( C_{5k5k} \) was certainly no larger than \( k^2 * C_{55} = 5k^2 \), it could have been much smaller.
Further, there were no illusions of improving upon exponential execution time: rather, the hope was that the base and constant terms in the exponential expression would prove to be so small that the barrier to larger boards would be machine address space rather than patience (or life expectancy).

The results obtained from executing this program, \( k = 1, 2, \ldots, 6 \), were better than had been hoped for: indeed, in each case we found \( C_{5s5s} = k^2 \cdot C_{ss} = 5k^2 \), and very little execution time was required: a least squares exponential fit to data showed execution time \((5k \times 5k \text{ board}) = 3.27 \cdot (2.123)^k \) seconds. But the most profound result was an 'accidental' discovery: there was a pattern for the \( 5 \times 5 \) board, call it \( P \), such that the configuration of Fig. 9(a) was a solution to the \( 15 \times 15 \) board! Thus \( P \) was a completely self-interfacing pattern, and we could obviously adjoin arbitrary strips of \( P \)-blocks to obtain any \( 5k_1 \times 5k_2 \) board. Hence \( C_{5k_15k_2} = k_1k_2C_{ss} = 5k_1k_2 \), for \( k_1, k_2 = 1, 2, \ldots \).

It was natural to ask: could this result extend to the other mod 5 congruence classes? A machine-based check showed that it did: for any pair of congruence representatives \( j_1, j_2 \in \{3, 4, 5, 6, 7\} \) there were patterns \( P_{j_15}, P_{j_1j_2}, P_{j_2}, \) and \( P_{55} \) for placing \( C_{j_15}, C_{j_1j_2}, C_{j_2}, \) and \( C_{55} \) queens in boards of the indicated sizes so that Fig. 9(b) was a solution to the \( (10 + j_1) \times (10 + j_2) \) board. Thus we could add (or subtract) an arbitrary number of identical strips to (from) the left edge or to (from) the bottom edge of Fig. 9(b) to obtain an arbitrary \( (5k_1 + j_1) \times (5k_2 + j_2) \) board with maximal queen placement.

In fact we have:

**Theorem 3.** For \( m, n \geq 3 \), \( C_{mn} = \lfloor \frac{1}{2}(m \cdot m) \rfloor \).

**Proof.** Writing \( m = 5k_1 + j_1 \) and \( n = 5k_2 + j_2 \), where \( j_1, j_2 \in \{3, 4, 5, 6, 7\} \), we have

\[
C_{mn} = 5k_1k_2 + k_2C_{j_15} + k_1C_{j_2} + C_{j_1j_2},
\]

but (calling on an elementary smallboard program) we know

\[
C_{j_2} = C_{5j_2} = j_2,
\]

so that

\[
C_{mn} = 5k_1k_2 + k_2j_1 + k_1j_2 + C_{j_1j_2} = (mn)/5 + (C_{j_1j_2} - \frac{1}{2}j_1j_2).
\]
Finally, calling on 'smallboard' again, we find $0 \leq C_{j_1,j_2} - \frac{1}{2} j_1 j_2 < 1$, giving the desired result. □

For the sake of completeness, we note that $C_{1,n} = \lceil \frac{3}{2}(1 \cdot n) \rceil$, and $C_{2,n} = \lceil \frac{3}{4}(2 \cdot n) \rceil$.

3.4. $N_{mn}$

There was yet another 'accidental' discovery made in running the program of Section 3.3: $N_{15,15} = N_{20,20} = N_{25,25} = N_{30,30} = 50$! This (the constancy) was indeed unexpected, but, upon viewing the individual solutions, the cause of the constancy, as well as a method for generalization to $N_{mn}$ became clear.

We need some additional notation. For each solution to an $m \times n$ board, let us define an associated pattern array as follows: first, index the solutions to each $j_1 \times j_2$ board, $j_1, j_2 \in \{3, 4, 5, 6, 7\}$ (this does require some straightforward machine computation with worst case 7×7: 5 minutes); then write $m = 5k_1 + j_1$, $n = 5k_2 + j_2$, and partition the $m \times n$ solution pattern into $(k_1 + 1) \times (k_2 + 1)$ blocks, as shown in Fig. 10. Replacing each pattern of CQ's with its associated index, we have a $(k_1 + 1) \times (k_2 + 1)$ array of pattern indices which conveniently represents the $m \times n$ solution.

![Diagram of pattern array](image)

**Fig. 10.**

**Theorem 4.** For every $r \geq 4$, for every $s \geq 4$, any pattern array of dimension $r \times s$ will have interior rows (rows other than first and last) which are all identical and interior columns which are all identical.

Thus, by removing interior rows and columns, we see that any placement of CQ's on a large board is necessarily a unique and trivial extension, obtained by replicating interior rows and/or columns, of a placement on a small board whose dimensions have the same mod 5 congruence classes as those of the large board.

**Proof of Theorem 4.** We use a nested induction: first consider the case $r = 4$. The theorem was verified for the case $s = 4$ by direct computer generation of all boards...
having each dimension between 18 and 22, inclusive. Since an efficient blockplacement program was used, the total execution time required here was roughly 7 minutes. Of course, the case \( r = 4, s = 4 \) chronologically preceded the theorem itself and was the basis of all the mathematical development.

Now assume the theorem is true for \( 4 \leq s \leq s_0 \), and consider a pattern array of dimension \( r \times (s_0 + 1) \) (Fig. 11). Removing first the leftmost column and then the rightmost column, we obtain two over-lapping pattern arrays of dimensions \( r \times s_0 \). By the inductive assumption, both have all interior columns identical, but since \( s_0 \) is at least 4, there is at least one column interior to both \( r \times s_0 \) boards; hence all interior columns of the \( r \times (s_0 + 1) \) board are identical. Finally, to see that the (two) interior rows are still identical, apply the inductive hypothesis of identical interior rows first to the left \( r \times s_0 \) sub-board and then to the right one (to catch the final column).

![Fig. 11.](image)

Now (moving to the outer loop) assume the theorem is true for the case \( 4 \leq r \leq r_0 \) (all \( s \geq 4 \)). Obviously an entirely parallel argument with the roles of rows and columns exchanged now completes the proof. □

We should say that we did push this (down) one dimension further: a complete generation of the pattern arrays of dimensions \( 3 \times 3 \), \( 3 \times 4 \), and \( 4 \times 3 \), shows that there is a one-to-one correspondence between each of these sets and the set of pattern arrays of dimension \( 4 \times 4 \), obtained, as expected, by insertion (deletion) of an identical interior row or column. A straightforward induction then extends us to the \( 3 \times s \) case, \( s > 4 \).

Thus we have established:

**Theorem 5.** For \( m, n \geq 13 \), \( N_{mn} \) is a constant which depends only upon the mod 5 congruence classes of \( m \) and \( n \), as given in Table 2.

Although constancy did empirically extend to lower dimensions within most of the congruence class pairs, it definitely does not extend in general! For example, consider \( N_{3,n} \) where \( n \equiv 5 \pmod{5} \). Each of the patterns of Fig. 12 will interface (end to end) with either itself or the other, giving us at least exponential growth in \( N_{3,n} \) as \( n \to +\infty \).

Strangely enough, the most empirically intriguing case left unsolved is \( 8 \times n \).
The crippled queen placement problem

Table 2

<table>
<thead>
<tr>
<th>n(mod 5)</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>24</td>
<td>4</td>
<td>46</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>44</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>m(mod 5)</td>
<td>5</td>
<td>46</td>
<td>44</td>
<td>50</td>
<td>52</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>20</td>
<td>50</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>10</td>
<td>52</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>

Fig. 12.

4. Comparisons

We have presented several approaches to the solution of a novel generalization of the 8-queens problem. In effect, we have tried to show not one 'best' solution, but a 'tree of choices' leading to a set of solutions with different properties. This tree of choices can be described by the outline shown in Table 3.

Table 3

<table>
<thead>
<tr>
<th>Run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x8</td>
</tr>
<tr>
<td>1. Exhaustive search</td>
</tr>
<tr>
<td>2. Branch and bound</td>
</tr>
<tr>
<td>3. Dynamic programming</td>
</tr>
<tr>
<td>(a) O(n(p_n)^3)</td>
</tr>
<tr>
<td>(b) O(nw_m)</td>
</tr>
<tr>
<td>4. Partitioned board</td>
</tr>
<tr>
<td>(a) 8x8</td>
</tr>
<tr>
<td>(b) 5k x 5k</td>
</tr>
<tr>
<td>(c) m x n; m, n &gt;= 13</td>
</tr>
</tbody>
</table>

We can (rather informally) also characterize the amount of time or thought (estimated by the individuals involved) expended on each solution, see Table 4.

In Table 4, * denotes the following: Some additional information was acquired as a result of previous computational experience with the problem by individuals who generated these solutions:
2: Experiments with solution 1 convinced these individuals that a faster version was desirable and suggested that considerable speed could be gained by eliminating partial solutions that could never be extended to maximal-CQ placement solutions.

3(a) Knowledge derived from one version of solution 1 showed that all 1-column solutions could be enumerated, and that only 28 of these exist for the 8 row by 1 column board. This suggested the value of the Dynamic Programming approach 3(a). Previous D.P. attempts were unsuccessful, but are counted as part of the solution-time spent. The actual time needed to generate the given D.P. solution, once the 1-column solution enumeration was known was less than ½ hour.

3(b) Required laborious work: about 16 hours beyond the work needed to derive solution 3(a).

4(b) This required solution of the 5 x 5 board as a sub-problem. In light of the method of 4(a), each was a rather straightforward programming exercise.

4(c) Most of the effort expended was in the development of a general-purpose conjecture-checker: a program which would solve (i.e. produce, through block placement, all pattern arrays) for the arbitrary m x n board. Necessary to this, of course, was a program to solve all 'small' j1 x j2 boards, j1, j2 E {3, 4, 5, 6, 7}.

Understandability and debugging effort expended in the actual development of these programs seemed to be quite similar. Everyone reported the expenditure of some 8–10 hours during program development and testing. All programs were fairly well-structured, in that they had simple flow-charts. Few were goto-less, and all used global variables. Several used few or no subroutines, instead consisting of a 2 or 3 page main program to perform the exhaustive search. Nonetheless, no substantial variation in debugging ease was noted. All programs occupied 4–7 pages of text, including documentation (which was nearly absent in some cases). Perhaps the small size of these programs makes the presence of good structuring and good documentation unimportant.

One conclusion can be drawn concerning solution 3(b): a substantial amount of program text, and almost all debugging effort, was expended in that portion of the program which computed certain sets (described in the Appendix and denoted there by W(r(R, S))) which were necessary to obtain the convenient indexing of the sets.
Since, in this version of the program, every debugging output statement which had to be inserted during testing remained present in the final version, the number of these statements gives a good indication of the debugging effort expended in each section of the program.

The following statistics can be derived from the final program text (all unlabelled measurements are in 'lines' of text):

**Solution (3)b:**

<table>
<thead>
<tr>
<th>Total size</th>
<th>Comments</th>
<th>Size, excl comments</th>
<th>Debugging statements</th>
<th>Number of procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total program</td>
<td>401</td>
<td>219</td>
<td>182</td>
<td>61</td>
</tr>
<tr>
<td>$W(r(R, S))$ module</td>
<td>325</td>
<td>192</td>
<td>133</td>
<td>49</td>
</tr>
<tr>
<td>I/O modules</td>
<td>11</td>
<td>0</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>D.P. algorithm</td>
<td>65</td>
<td>27</td>
<td>38</td>
<td>12</td>
</tr>
</tbody>
</table>

Taking ratios, we find that the following percentages of lines are devoted in each module to comments, code, and debugging.

**Solution 3(b):**

<table>
<thead>
<tr>
<th></th>
<th>% comments</th>
<th>% code</th>
<th>% debugging</th>
<th>% of total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire program</td>
<td>54.6</td>
<td>30.2</td>
<td>15.2</td>
<td>81.0</td>
</tr>
<tr>
<td>$W$ module</td>
<td>59.0</td>
<td>25.8</td>
<td>15.1</td>
<td>81.0</td>
</tr>
<tr>
<td>I/O module</td>
<td>0</td>
<td>100.0</td>
<td>0</td>
<td>2.7</td>
</tr>
<tr>
<td>D.P. module</td>
<td>41.5</td>
<td>40.0</td>
<td>18.5</td>
<td>16.2</td>
</tr>
</tbody>
</table>

Percent of non-comment statements devoted to debugging:

**Solution 3(b):**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire program</td>
<td>33.5</td>
</tr>
<tr>
<td>$W$ module</td>
<td>36.8</td>
</tr>
<tr>
<td>I/O module</td>
<td>0</td>
</tr>
<tr>
<td>D.P. module</td>
<td>31.6</td>
</tr>
</tbody>
</table>

We conclude that debugging statements, and probably debugging effort, is a linear function of the total number of statements in all modules, amounting to about $\frac{1}{3}$ of the total statements. However, the complexity of the $W$ module accounts for $\frac{2}{3}$ of the code, and a corresponding share of the debugging effort. Probably 4 times as much time and effort as was needed for the Dynamic Programming module was used in developing the $W$-module. Quite probably, the speedup of a factor of 23 gained here at the expense of some 16 hours of development effort, and about the same number of hours of debugging effort was unjustified by this performance.
improvement. On the other hand, no data on the effort needed to develop a more straightforward implementation of the D.P. algorithm, running in time $O(p^3_m)$, is available. (It could be estimated at about $\frac{1}{3}$ the time and effort needed for development of the current $W$-module.)

For comparison, we present program size and complexity statistics for the fastest implementation of solution 2 (a program written by Charles Poirier):

**Solution 2:**

<table>
<thead>
<tr>
<th></th>
<th>Total size</th>
<th>Comments</th>
<th>Size, excl comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total program</td>
<td>168</td>
<td>93</td>
<td>75</td>
</tr>
<tr>
<td>I/O module</td>
<td>30</td>
<td>4</td>
<td>26</td>
</tr>
<tr>
<td>Main algorithm</td>
<td>138</td>
<td>89</td>
<td>49</td>
</tr>
</tbody>
</table>

This program includes no procedures, and no debugging statements. Its text is also complicated by the presence of 3 goto statements. We note that, when debugging statements and I/O-related statements are excluded from solution 3(b)'s statistics, we get the following comparison:

Solution 3(b): 110 lines, 84 in $W$ module,

Solution 2: 49 lines.

Had the $W$ module been replaced by a slower version to implement solution 3(a), occupying about 30 lines of code, the comparison would be:

Solution 3(a): 56,

Solution 2: 49.

Thus, the programs would be of approximately the same size, and of approximately the same complexity. The size of solution 3(b) is in large part due to its complex $W$ module, and this added complexity results from a choice made at the 'second level' of the choice tree.

Of all the solutions to the $8 \times 8$ board, 4(a) required the fewest lines of code and the least amount of execution time. Nevertheless, when development time and requisite documentation, including theorem statements and proofs, are added, the solution becomes one of the longest and most time-consuming of all, and thus, as a 'stand-alone' effort, 4(a) is not an unqualified success. Of course, this is somewhat unfair in that 4(a) ultimately led to the closed-form solutions for $C_{mn}$ (all $m, n$) and $N_{mn} (m, n \geq 13)$, and the value of these solutions must also be weighed when making the comparisons.

We can, for this problem, document the fact that effort expended at the outermost level of the program development hierarchy, namely, while deciding on an overall approach to the problem, produces the greatest improvement in program performance. Furthermore, such effort seems to leave program complexity unaffected.
Although the resulting program may operate using principles that are not apparent, and which need proof, the actual code produced uses these principles in straightforward fashion.

5. Conclusions

We have presented in this paper a fairly detailed description of the design process as it applies to the solution of one problem. In the course of the solution, the designer must make choices at several levels of detail. Initially, a choice of solution strategy is needed. We have shown the initial cost differences incurred in making this strategic choice:

1. Straightforward, in analogy with 8 queens: 240 minutes,
2. Branch and bound: 4.8 seconds,
3. Dynamic programming: 0.6 seconds,
4. Board partitioning: 0.2 seconds.

Other measures of solution quality including program complexity are either essentially constant, or improve in order from choice 1 through choice 4.

However, our best attempts to guide further choices at lower levels of detail via estimates of the improvements to be gained have to be judged failures:

Choice 1 vs 2: No easy estimate of the improvement to be expected as a result of a branch-and-bound solution was possible. It was far easier to make the minor change in solution 1 that produced solution 2, and see what happened than to perform any a priori mathematical analysis.

Choice 3a vs 3b: Initially, we thought that relatively few triples of adjacent column-patterns were mutually compatible. We had no notion of how large $W_m$ was. The mathematical analysis presented in the Appendix was carried out in part to determine this quantity. Once completed, we decided to develop the $O(n \times w_m)$ algorithm (which was fairly straightforward once the analysis giving enumeration of $w_m$ had been done). Yet the effort expended to obtain Table 1 was unjustified by the linear factor of 23 improvement we gained.

Choice 4: A priori estimates of the gains to be made at each stage of the development of solution 4 were, at best, of a probabilistic and subjective nature. Although the derivation of a closed-form solution remained feasible at each stage of the development, the possibility of no return (other than insight into the complexity of the problem) on time invested also remained. The chief merit of solution 4 (all parts) is that it is representative of a research technique which we feel will become increasingly important in both mathematics and computer science: machine-generation of theorem statements (the 'accidents' of Section 3), followed by their inductive proofs.
We hope that documenting the issues which arose during the design of one non-trivial program will prove valuable to researchers in program methodology. They may be of value also to designers of Computer Science curricula. We feel that this study shows the value of a general background in mathematical problem solving when attacking novel problems.

When mathematical reasoning is coupled with use of the computer to resolve conjectures, a powerful problem solving mode results. This produces high-quality software products.

Appendix A. An indexing of the set of compatible column-pattern triples

Solution 3 requires indexing over the set \( W_m = \{(R, S, T): (R|S) \) and \( (S|T) \) and \( (R \parallel T) \}. \) Here we show how to compute a convenient bijection \( h: I_{w_m} \rightarrow W_m \) where \( I_k = \{0, 1, \ldots, k-1\} \) and \( w_m = |W_m| \), the number of objects in the set \( W_m \). The techniques used to develop this bijection can be exploited in two ways:

1. We can easily develop a tabulation technique that, for any \( m \), gives \( W_m \); and
2. We can design computer programs that compute the sets \( W(r(R, S)) = \{r(S, T): (R, S, T) \in W_m\} \), where \( r: X_m \rightarrow I_{w_m} \) is a bijection with domain \( I_{x_m} = |X_m| \).

Thus, the indexing over all \( T \) such that \( (R, S, T) \in W_m \) is reduced to indexing over all elements of \( W(r(R, S)) \).

A.1. The set \( X_m = \{(R, S): R|S\} \)

We proceed to derive recursive formulas for the number of solutions to an \( i \times 2 \) board, for \( i = 1, \ldots, m \). Let \( L0(i) \) be the number of different legal CQ placements possible on the \( i \times 2 \) board, without inter-CQ attack. Consider row \( i \) of the board. Legal CQ placements include some with no CQ’s in row \( i \), others with a CQ in column 1, or in column 2, but none with a CQ in both columns.

Let \( P0(i) \) be the number of solutions to the \( i \times 2 \) board with no CQ’s in row \( i \), \( Pj(i) \) be the number of solutions to the \( i \times 2 \) board with a CQ in row \( i \) column \( j \), for \( j \in \{1, 2\} \). Then:

\[
L0(i) = P0(i) + P1(i) + P2(i),
\]

\[
P0(i) = L0(i-1) \quad \text{and} \quad P1(i) = P2(i).
\]

To develop a recursive formula for \( P1(i) \), note that a CQ in row \( i \) column 1 attacks squares so that the squares which remain available for CQ occupancy form a 2 column board whose 1st column contains \( i-3 \) rows, and whose second contains \( i-2 \) (See Fig. A1). Let \( L1(i) \) be the number of legal CQ solutions possible on a 2-column board, one of whose columns is \( i \) rows long, the other \( i-1 \) rows long. Then \( P1(i) = P2(i) = L1(i-2) \).
Now \( L1(i) \) can be analyzed similarly: its \( i \)th row either holds no CQ, giving \( L0(i - 1) \) solutions, or it holds one CQ, giving \( P1(i) = P2(i) = L1(i - 2) \) solutions. Thus, we are led to the following formulation:

\[
\begin{align*}
L0(i) &= L0(i - 1) + 2L1(i - 2), \\
L1(i) &= L0(i - 1) + L1(i - 2),
\end{align*}
\]

and thus

\[
L0(i) = L1(i) + L1(i - 2).
\]

We note that \( L0(0) = 1 \) (There is one solution—the empty board—for the board with no rows), and \( L1(0) = 1 \) for the same reason. Similarly, the reader can verify directly that \( L0(1) = 3 \) and \( L1(1) = 2 \). Using (A.1) and (A.2) we can then generate Table A1. Thus \( x_8 = 193 \), which is smaller than \( \frac{1}{4} \times (p(8))^2 \), since \( p(8) = 28 \) = the number of \( 8 \times 1 \) solutions.

<table>
<thead>
<tr>
<th>( i )</th>
<th>( L1(i) )</th>
<th>( L0(i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>44</td>
<td>57</td>
</tr>
<tr>
<td>7</td>
<td>81</td>
<td>105</td>
</tr>
<tr>
<td>8</td>
<td>149</td>
<td>193</td>
</tr>
</tbody>
</table>

Note that closed-form expressions for \( L0(i) \) and \( L1(i) \) can be developed: by substituting (A.2) into (A.1) we obtain \( L1(i) = L1(i - 1) + L1(i - 2) + L1(i - 3) \), and similarly by combining (A.1) and (A.2) and substituting into (A.0) we obtain \( L0(i) = L0(i - 1) + L0(i - 2) + L0(i - 3) \). Each linear difference equation thus has characteristic polynomial \( X^3 - X^2 - X - 1 \) whose 3 roots, \( S_1 \), \( S_2 \), and \( S_3 \), can be expressed in terms of radicals. The general closed-form expression is then \( L0(i) = C_1 S_1^i + C_2 S_2^i + C_3 S_3^i \) where the coefficients are determined by the initial values.
course, the same is true for $L1(i)$.) In particular, since the real root is approximately 1.839286 and the two complex roots have an approximate norm of 0.7373527, we can say that $x_m \approx O(1.84^m)$.

To develop the function $r$, we first describe a simple representation of the 2-column solutions by means of strings of digits. Each 2-column solution has the property that its $i$th row contains either 0 or 1 CQ's. Let $S$ be a string of digits over the alphabet $\{0, 1, 2\}$. Then $S$ describes an $|S| \times 2$ pattern of CQ placements, by the correspondence: row $i$ of the solution contains one CQ in column $j$ ($j = 1$ or 2) if the $i$th digit of $S$, $S(i)$, satisfies $S(i) = j$.

If $S(i) = 0$, the solution corresponding to string $S$ has no CQ in row $i$.

Now, some strings $S$ as above describe legal solutions, while others do not. Let us define the function $f$ on strings $S$ as follows:

$$f(S) = \text{the number of legal solution strings } T \text{ which are lexicographically ordered less than } S, \text{ if } S \text{ is a legal solution string; undefined, otherwise.}$$

Then

$$f(0S) = f(S),$$
$$f(10S) = L0(|S| + 1) + g(1, S),$$
$$f(20S) = L0(|S| + 1) + P1(|S| + 2) + g(2, S) = L0(|S| + 1) + L1(|S|) + g(2, S),$$
$$f(xys) = \text{undefined otherwise,}$$

where, for $i = 1, 2$, $g(i, S)$ denotes the number of strings which precede $S$ on a board with $|S|$ rows which has one square removed from column $i$, row $|S|$. So

$$g(i, 0S) = f(S),$$
$$g(3 - i, iS) = L0(|S| + f(S)),$$
$$g(i, yS) = \text{undefined otherwise.}$$

Furthermore,

$$f(0) = 0 = f(\lambda) \text{ where } \lambda = \text{the null string},$$
$$f(1) = 1, \quad f(2) = 2.$$

The first few solution strings, and their corresponding $f$ values can be tabulated using Table A1, see Table A2.

Each solution string $S$ can be preceded by any number of 0 digits to modify it to describe solutions for values of the row index $i > |S|$. Since $f(0S) = f(S)$, the value of $f(S)$ is independent of $i$, so long as $i \geq |S|$. Furthermore, a deterministic finite state machine can be designed to read string $S$ (from left to right), determine if $S$ represents a legal solution, and develop the value of $f(S)$ (see Fig. A2).

Note that we can now define $r$: if $(R, S) \in X_m$, then we can write $R = (R_m, R_{m-1}, \ldots, R_1)$ where each $R_i = 2$, if a CQ appears in row $i$, and $R_i = 0$, otherwise. Similarly $S = (S_m, S_{m-1}, \ldots, S_1)$ where $S_i = 1$, if a CQ appears in row $i$, and $S_i = 0$ otherwise. Then $r(R, S) = f((R_m + S_m)(R_{m-1} + S_{m-1}) \cdots (R_1 + S_1))$. 

Table A2

<table>
<thead>
<tr>
<th>String $S$</th>
<th>$f(S)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>102</td>
<td>6</td>
</tr>
<tr>
<td>200</td>
<td>7</td>
</tr>
<tr>
<td>201</td>
<td>8</td>
</tr>
<tr>
<td>1000</td>
<td>9</td>
</tr>
</tbody>
</table>

Fig. A2. A deterministic finite-state machine to calculate $F(S)$: start state: 0. Initially $Q=0$; on termination $Q$ holds $F(S)$.—A: add $10(J-1)$ to $Q$. B: add $L1(J)$ to $Q$, where $J$ is the number of digits between the read-head and right end of string, inclusive.

A.2. The set $W_m = \{(R, S, T) : R|S, R|T, S|T\}$

We begin in much the same way as in Section A.1 to develop the number of solutions for a 3-column by $i$-row board. Somewhat more complexity results, since more CQ placements are possible in row $i$, and in addition, more distinct board shapes must be considered. See Fig. A3.

Fig. A3. 3-column by $i$-row board shapes. Status of board squares are shown by symbols as follows: $Q$=occupied, $-$=attacked or off board, $x$=available.
Let $L_3s(i)$ be the number of legal CQ placements for a board with $i$ rows and 3 columns, in which certain squares in rows $i$ and $i-1$ are not 'available' for CQ placement. Specifically:

<table>
<thead>
<tr>
<th>$s$</th>
<th>available in row</th>
<th>$i$</th>
<th>$i-1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>all</td>
<td>all</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>all</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1,3</td>
<td>all</td>
<td></td>
</tr>
</tbody>
</table>

Then, from Fig. A3,

$$L_30(i) = L_30(i-1) + L_33(i-1) + L_313(i-2) + L_31(i-1),$$

where the summands correspond respectively to the case where there are 0 CQ's in row $i$, or one CQ in columns 1, 2, or 3 of row $i$. Now $L_33(i) = L_31(i) = L_32(i-1) + L_32(i-2)$, where the summands correspond to 0 or 1 CQ in the only available square of row $i$. Similarly, $L_32(i) = L_30(i-1) + L_313(i-2)$, and $L_313(i) = L_30(i-1) + L_33(i-1) + L_31(i-1)$, for no CQ's or a CQ in columns 1 or 3 respectively. These equations can then be reduced to:

$$L_30(i) = L_30(i-1) + 3L_30(i-3) + 4L_30(i-4) + 4L_30(i-5) + 2L_30(i-6).$$

Now the characteristic polynomial $y^6 - y^5 - 3y^3 - 4y^2 - 4y - 2$ has real roots 2.202044 and −0.667283 (approx.) and two pairs of complex roots with norms 0.8678813 and 1.34427 (approx.). Thus, as with $x_m$ earlier, we can say $w_m = L_30(m) = 0(2.21^m)$. (See also Table A3.)

<table>
<thead>
<tr>
<th>$i$</th>
<th>$L_313(i)$</th>
<th>$L_30(i)$</th>
<th>$L_33(i)$</th>
<th>$L_32(i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>9</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>18</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>40</td>
<td>17</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>74</td>
<td>89</td>
<td>38</td>
<td>55</td>
</tr>
<tr>
<td>6</td>
<td>165</td>
<td>197</td>
<td>81</td>
<td>121</td>
</tr>
<tr>
<td>7</td>
<td>359</td>
<td>433</td>
<td>176</td>
<td>271</td>
</tr>
<tr>
<td>8</td>
<td>785</td>
<td>950</td>
<td>392</td>
<td>598</td>
</tr>
</tbody>
</table>

This analysis, conducted before proceeding with the development of algorithms to enumerate $W_m$, showed:

(1) For $m=8$, $w_m = 950$, which is proportional to the execution time of solution 3 if $W_m$ is enumerated. The corresponding time of $p(8)^3 = 28^3 = 21952$ is a factor of more than 23 greater. Thus it seemed worthwhile to proceed with the development, presented in Section A.1, of the finite state machine used to compute the function $f(S)$, where $S$ represents any $|S| \times 2$ solution.
(2) Solution 3 will require time $O(n \times 2.21^m)$ to solve the general $m$ row by $n$ column CQ placement problem.

A.3. An algorithm to compute the sets $W(r(R, S)) = \{r(S, T): (R, S, T) \in W_m\}$

The general approach is to generate strings of digits over \{0, 1, 2, 3\} representing solutions to the $i \times 3$ board, using a collection of recursive subroutines. As each digit $S_i$ of a string $S$ is generated, $S_i$ is translated (by indexing into a table) into a pair $(T_{i1}, T_{i2})$ of digits, representing $S$ as two over-lapping 2-column solutions. Specifically, $S_i > 0$ indicates the presence in $S$ of a CQ in column $S_n$, row $i$. The corresponding $T_{i1}$ value is 0 if $S_i=0$ or $S_i=3$, and $T_{i1}=S_i$ otherwise. Similarly, $T_{i2}$ is 0 if $S_i=0$ or $S_i=1$, and $T_{i2}=S_i-1$ otherwise. The sequences $(T_{i1})$ and $(T_{i2})$ can then be fed into the finite state machine presented in Section A.1 to produce the integer pair sequence $(f(T_{i1}), f(T_{i2}))$. When string $S$ represents the member $(S_1, S_2, S_3)$ of $W_m$, $f(T_{i1})=r((S_1, S_2))$ and $f(T_{i2})=r((S_2, S_3))$.

In practice, the strings $S$ are generated, digit by digit, without backtracking, so that whenever a string $S$ has been produced it is known that there is some suffix $Q$ of digits such that $SQ \in W_m$. This suggests that the digits $T_{i1}$ and $T_{i2}$ be presented in parallel to two separate copies of the $f$-computing FSM. Whenever $|S|=m$, the values computed thus far by each FSM copy are used as $f(T_{i1})$ and $f(T_{i2})$ respectively. A record is kept of the state of each FSM (including its value of $f$), so that digits of a $T$-string can be removed, and a new suffix substituted for them, without recomputing $f$ on the unchanged prefix of the $T$-string. Our $W$-enumeration algorithm then pushes $f(T_{i2})$ onto queue $W[f(T_{i1})]$, after initializing all queues $W[i]$ empty.

There remains the mathematical problem of producing recursive subroutines to generate strings $S \in W_m$, without backtracking. This problem was solved by examining the recursive equations for $L3s(i)$, together with their derivations. One recursive routine, $TRY3s(i)$, was built for each shape $s$. The job of routine $TRY3s(i)$ is to produce every solution to the shape $s$ board with $i$ rows. Each digit $S_i$ produced is fed, along with $i$, to routine $Q(S_n, i)$, which translates $S_i$ into $T_{i1}$ and $T_{i2}$, and feeds it as digit $m-i+1$ of an $m$ digit string $(T_n)$ to FSM copy $k$. Routine $TRY3s(i)$ produces the first legal value $V_i$ of digit $S_n$ and then calls another $TRY3s'(i-J(V_i))$ routine to produce all solutions having initial digit $S_i=V_i$ before producing any solutions with $S_i > V_i$. (Here, $J(V_i) \in \{1, 2, 3\}$. $J(V_i)$ is generated along with the value $V_i$.)

For example, routine $TRY313(i)$ is:

```c
if (i>0) {
    Q(0, i); TRY30(i-1);
    Q(1, i); Q(0, i-1); TRY32(i-2); Q(3, i-1); Q(0, i-2); TRY32(i-3);
    Q(3, i); Q(0, i-1); TRY32(i-2); Q(1, i-1); Q(0, i-2); TRY32(i-3); 
} else soln(); return;
```
Routine soln() is responsible for updating queues W[i], and recording the number of CQ's in columns 1 and 2. Routine Q keeps track of the number of CQ's placed into columns 1 and 2 by each solution-string S. Communication among routines Q() and soln() takes place using global variables, and the states of FSM copy j are recorded in a statically allocated global array QM at QM[i][j], so that the replacement of a suffix of a 2-column string by a different suffix is simple, for all earlier states of each FSM copy are accessible to routine Q(). This recursive technique produces a constant number of subroutine calls for each digit of every solution-string it produces. However, it produces a tree T of subroutine calls whose leaves represent solution strings by the correspondence: The solution represented by leaf L is given by the sequence of labels of tree arcs on the unique path from the tree root to L, where each tree arc is labelled by a digit. This tree of solution strings has no more than 2 * w_m arcs, total, for it has at most w_m internal nodes, where w_m is the number of leaves of T. The entire solution S includes computation of the sets W(r(R, S)), thus requires time O(w_m * (n+1)) to solve the m row, n column CQ placement problem.

References