BIT 12 (1972), 347–365

AN EXPERIMENT ON PROGRAM DEVELOPMENT

PETER NAUR

Abstract.

As a contribution to programming methodology, the paper contains a detailed, step-by-step account of the considerations leading to a program for solving the 8-queens problem. The experience is related to the method of stepwise refinement and to general problem solving techniques.

Key words: programming methodology, education in programming, programming techniques, action clusters.

1. Introduction.

In the recent papers E. W. Dijkstra [1] and N. Wirth [6] have discussed the train of thought going into the gradual development of programs as solutions of given problems. In both of these papers considerable stress is put on a development process in which successively "more primitive" actions that achieve the desired net effect are considered. Wirth considers the creative activity of programming as a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures, or a process of successive refinement of specifications.

While the importance of the quoted contributions to our understanding of the process of program development is evident, still the question may be raised whether this approach to program development is the proper ideal to be set. Comparing with the discussion in the Garmisch conference on Software Engineering [5] (see particularly pages 47 to 48), the method advocated by Dijkstra and Wirth must be classified as a strict top-down approach. However, as expressed by B. Randell, S. Gill, and R. Barton, such an approach cannot be expected to work, at least not for the large software systems considered at the Garmisch conference.

As a second point of uncertainty, Dijkstra and Wirth, although certainly aware of the potentialities of proofs of programs, do not, in their descriptions of the programs being developed, include direct proofs, or generally valid descriptions of state variables, like those advocated by the present writer [4].

Further, as a third point, Dijkstra and Wirth, although concerned with methodology of program development, do not relate their principles to

Received March 4, 1972.

existing work on the psychology of problem solving, as described, e.g. by Hyman and Anderson [2]. Thus, as one example, the insistence of Dijkstra and Wirth on step-wise refinement could well get in conflict with the first two problem solving precepts of Hyman and Anderson: Precept I: "Run over the elements of the problem in rapid succession several times, until a pattern emerges which encompasses all these elements simultaneously"; precept II: "Suspend judgement. Don't jump to conclusions".

In view of the above considerations it seems that more work on the problem is needed. One area of work would seem to be direct observation of what actually happens during realistic program development, in order to allow empirical studies. This is where the present report comes in. It was initiated by the suggestion at the end of section 1 of the paper by Wirth [6] that "the reader is strongly urged to try to find a solution by himself before embarking on the paper ...". Looking at the formulation of the problem (the 8-queens problem) and finding that it was not one which I had done before, I decided to follow this suggestion and to make the solution a demonstration of the way in which I would attack the problem, before reading the rest of the paper. For the purpose of documentation I decided to record, as far as possible, my complete train of thought as typed notes. This mode of work is one I have used frequently in the past. Where necessary the notes were supplemented by handwritten drawings and program pieces.

The following description is the outcome of this program of work. The text given is practically identical with the one which I typed during the development of the program. For easier reference during the work I made a section break, starting with a section number, each time I made a shift of attention. Notes added later are enclosed within square brackets: [Note]. The initial work up to and including point 36 was carried out over a period of about one month, in competition with many other tasks. The total time used must have been of the order of 15 hours.

8-queens problem (Wirth [6]):
Given are an 8 × 8 chess board and 8 queens which are hostile to each other. Find a position for each queen (a configuration) such that no queen may be taken by any other queen (i.e. such that every row, column, and diagonal contains at most one queen).

2. Eight queens - thinking at the typewriter.

1

Queens can move up and down, right and left, and diagonally. A crucial point is the choice of the representation of configuration (used here with 0, 1, 2, ... 8 queens, not only with 8) such that the possibility of mutual captures is easily determined by an algorithm.

2

It seems highly likely that the chessboard must be represented as an array. There are several possibilities:

integer array *ROW*[1:8]; **comment** *The value indicates either not occupied* −0, *or the column where a queen is*;
Boolean array *BOARD*[1:8, 1:8]; **comment** *not occupied* — **false**, *occupied* — **true**;

Still further possibilities are conceivable, e.g. a vector corresponding to each diagonal. Somehow they do not seem reasonable.

3

How to find a configuration? I do not see much further than simple trial and backtracking in some systematic order which makes sure to cover all possibilities. Still, for efficiency, the best systematic order of trial may be important. Because of the diagonal coverage, the middle fields exclude more other fields than the rest, so we should presumably start there. Another important point: symmetry! It will save a good deal of time to avoid finding symmetric solutions more than once. Idea: start with the first queen or two in some way which excludes symmetric solutions (is this necessarily possible? — I don't really know right now). [It is clear that I am planning to find all solutions, and not just one as required in Wirth's statement of the problem].

4

More on symmetry and method: Clearly, each solution has just one queen in each row, and just one in each column, because of the capture along rows and columns. Think, for example of column 4 (Fig. 1), this

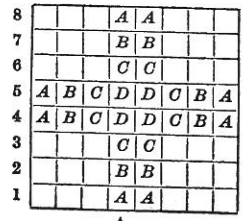

Fig. 1.

because it is likely to be a good one to start with. A configuration having a queen in row 1 of that will be symmetric with 7 other configurations indicated as *A*. Similarly with *B* and *C*. *D* only occurs in 4 symmetric solutions. Consequently, to avoid symmetric solutions, if a queen is tried in each of rows 1, 2, 3, and 4 of column 4, then it should not be started in any other of the fields marked *A*, *B*, *C*, *D*, except when two queens can be placed in two positions having the same letter, or three, or ... — Queer, this is hard, let it wait for the moment.

5

Representation and method by trial and backtracking: There seems to be two ways (or just tendencies?) to handle a trial: Either we just record the configurations, with the minimum amount of data to make it unique (approach P). In approach P, to check whether the next queen placed is admissible we have to compare its position with that of all queens placed already. Or, approach Q, we record not only the minimum description of the configuration, but a number of immediate consequences of it, which make the following check for the placing of a queen simpler. Example of approaches P and Q: Approach P would go well with representation as *ROW* in point 2. In approach Q the array *BOARD* might record not only queens, but also fields under attack.

6

Choice of P or Q: P is simpler in backtracking, because no consequences need to be removed, but it is harder in each trial, because it may take longer to go through the check process when trying to place one queen. Approach P seems easier to do right. The relative efficiency is hard to compare without more details. For the present, I tend to P.

7

Representations and the given requirements: The choice of a representation may imply that certain given requirements are automatically fulfilled. Thus point 2, *ROW*, excludes that a column has more than one queen.

8

New idea for a representation:

integer array *QUEENROW*, *QUEENCOLUMN*[1:8]; **Comment** *Gives where each queen is now placed.*

If this is sorted according to the values of *QUEENROW*, the corresponding elements of *QUEENCOLUMN* are the same as *ROW* of point

2. It does not seem much good, primarily because it introduces a numbering of the queens, which is of no use or significance.

9

Yet another idea for a representation, inspired by the symmetry of point 8: Add to *ROW* of point 2 a similar *COLUMN*, telling for each

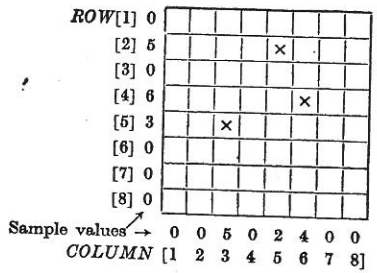

Fig. 2.

column the number of the row where a queen is placed (Fig. 2). This is a device like what is needed in approach Q of point 5. With approach P it does not seem to help much.

10

Two different needs for a representation: 1) Knowing whether a field is already occupied, and 2) knowing whether a field is under attack. These two needs do not necessarily have to be served by two different mechanisms.

11

New idea on method: Induction: Would it be reasonable to let the trial solutions develop through boards of 2,3,... fields square? Suppose we have a solution for a 7×7 board: the remaining problem is to place either two, or just one queen (at the diagonal). I.e. the 7-board must have 7 or 6 queens. It looks complicated!

12

Summary: I cannot see easily through the problem of symmetry. I think I will just confine myself to reducing some of it away, and then find what remains in the resulting configurations. Second: I choose the simple *ROW* representation—the others don't seem to gain much that I can see. Third: For efficiency and to reduce the symmetry I place the two queens

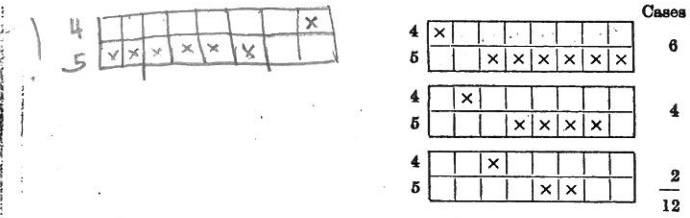

Fig. 3. Starts at rows 4 and 5.

at rows 4 and 5 by hand, see Fig. 3. Unless I cheat myself there are only 12 starting positions, instead of 8×8, pretty good! Fourth: As a modest contribution to further efficiency I draw the column numbers of the remaining six queens from a vector of the remaining numbers, after the rows 4 and 5 have been taken care of. Fifth: To further cut down the branches of the search tree early I add the following rows from the middle in the order: 3, 2, 1, 6, 7, 8.

13

The algorithm for selecting the starting values in rows 4 and 5 is given in Fig. 4.

```
for c4 := 1, 2, 3 do
begin ROW[4] := c4;
  for c5 := c4+2 step 1 until 9−c4 do
  begin ROW[5] := c5;
    q := 1;
    for k := 1 step 1 until c4−1, c4+1 step 1 until c5−1,
      c5+1 step 1 until 8 do
      begin Q[q] := k; q := q+1 end;
```

Fig. 4.

14

The algorithm for making the table of the remaining 6 column numbers can be made in several ways: 1) start filling in up to 8 and then shift down at c4 and c5. 2) Make real use of the crazy possibilities of the Algol 60 for statement, see Fig. 4. Here we make use of the fact that $c4 < c5$.

15

The algorithm for the final test of a new queen just added to the board has been at the back of my mind ever since the start. It must run something like:

```
row := some new row number; col := some queen column number;
for k := lowrow step 1 until row−1, row+1 step 1 until highrow do
begin qu := ROW[k];
if qu = col then go to reject;
if abs(row−k) = abs(qu−col) then go to reject
end;
```

16

New idea, revision of 14 and 15: The columns may be represented as a **Boolean array** *FREE*[1:8] telling whether that column is still not covered by a queen. This will make for a much more efficient search, since attack on the column is detected right away. Thus the test on $qu = col$ in 15 is unnecessary. *FREE* must be initialized to be all **true**. Right after setting the values of c4 and c5 we set: *FREE*[c4] := *FREE*[c5] := **false**—rather c4 is set just inside the c4 loop, c5 just inside its loop, and set back to **true** again at the end of their loops.

17

Time for an overall view of the program structure, see Fig 5. Declarations—c4 loop with settings of *FREE*, same for c5—Initialize tables. Remark: to implement the rather queer order of taking in new rows, and to calculate the corresponding *lowrow* and *highrow* for point 15 I plan to use tables of six entries, corresponding to the last six queens. For the rest, there is a problem to get things into an order which is sufficiently transparent to be understood. There will be a "master pointer", *m*, taking values up and down from 1 to 6, telling which queen, beyond the first two, is being played with. It's time to establish some action clusters! (Cf. [4]).

Fig. 5.

```
Declarations
Initialize tables of free, lowrow, row, highrow
for c4
begin free[c4] := false
  for c5
  begin free[c5] := false
        m := 0;
    P: Start to consider new row
```

```
        m := m+1; low—row—high; ROW[row] := 0;
    Q: Select next possible column
        if there are no more then go to reject row;
        Record column selected
        Test for diagonal capture—point 15
        if then go to reject column
    Success:
        if less than 8 queens then go to P;
        Print configuration;
    reject column:
        Clear record of column; go to Q
    reject row: decrease m;
        if m > 0 then go to Q;
    free[c5] := true
  end
  free[c4] := true
end
```

... Break of several weeks ...

18

Comment on the general attitude: I have not tried to get to some elegant solution, but rather tried to show the interplay of alternative possibilities in various directions. The stress on efficiency is perhaps unreasonable in this particular problem. It is high time to try some quantitative estimates on the search time.

19

Estimates of search time: In a very primitive strategy we take each of the 8! = ca. 40 400 permutations and try each of them. Guess at time to produce one permutation: 16 times taking a subscripted variable (minimum time is 8 times a subscripted variable). Guess at the average time to reject a permutation: let us say that we have to combine rows 1, 2, and 3 with the rest to find a conflict, i.e. $7 \times 6 \times 5$ pairs = 210 pairs of queens, each requiring one subscripted variable plus something more, let us say two subscripted variables. Total for one permutation: $16 + 2 \times 210 = 436$ subscripted variables. In Gier: 436×0.7 ms = 0.3 sec. For all permutations: 12000 sec. = 3.3 hours. That is too much, so there is good reason to do something about it. The approach of point 12 gives $12 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$ permutations = 8640, i.e. a factor of 4.6 better. If again we guess that three more queens must be tried, on the average, to find a conflict, we get the number of pairs to check: $3 \times 4 \times 5 = 60$ pairs to check, and $6 \times 5 \times 4 = 120$ part positions to generate. If each check takes 2 and each position takes 6 subscriptions, we require $60 \times 2 + 120 \times 6 = 120 + 720 = 840$ subscriptions = 0.6 sec. This is for each of the 12 starting positions, so in total 12×0.6 sec = 7.2 sec! This surely is wrong, why is it so much less than 3 hours/factor 4.6 = 2600 sec? Or perhaps it is just the effect of cutting off the branches of the tree without even looking at the leaves. If so there is not much reason to try to go further in the attempt to get efficiency.

Very interesting

20

A start on the details of the description of the situation and the action clusters to go with it:

(1) m goes from 0 to 6 and tells which queen, beyond those on rows 4 and 5, is being played with, according to the following scheme:

m	Occupation of rows: . not occupied, x occupied, not last, l last occupied								*lowrow*	*highrow*	*row*	Revised according to point 30		
	1	2	3	4	5	6	7	8				*lowrow*	*highrow*	*row*
0	.	.	.	l	x	.	.	.	4	5	4	5	5	4
1	.	.	l	x	x	.	.	.	3	5	3	4	5	3
2	.	l	x	x	x	.	.	.	2	5	2	3	5	2
3	l	x	x	x	x	.	.	.	1	5	1	2	5	1
4	x	x	x	x	x	l	.	.	1	6	6	1	5	6
5	x	x	x	x	x	x	l	.	1	7	7	1	6	7
6	x	x	x	x	x	x	x	l	1	8	8	1	7	8

(2) Further general statements: When $m > 0$ the situation of queens corresponding to the rows from *lowrow* to *highrow* for $m-1$ [revised acc. to point 30: m] form a possible position, in other words, if the queen placed in the row given by the value of *row* is removed, the position becomes free of attacks.

Further concepts, valid for a given value of $m > 0$: The *current row*: the same as the last one to be occupied, indicated by letter l, and in the column *row*; *previous rows*: the current rows for all m smaller than the one given, plus rows 4 and 5.

(3) Further general statement: For each previous row p the column number of a queen is given by $ROW[p]$ ($= 1, 2, \ldots,$ or 8). For each such column number we have $FREE[ROW[p]]$ = **false**. For the current row we have $col = ROW[row]$ either = 0, or equal to a column number not

used by previous rows. In the latter case: $FREE[ROW[row]] = \text{false}$. All other elements of *FREE* are true.

(4) Further general statement: for a given value of m and a value of *col*, all trees having the same positions of queens in all previous rows, and with a queen placed in all columns, $1,2,\ldots col-1$, have been sufficiently tried.

Cluster 1: increase m.

Conditions for use: $m < 6$, the queen placed in the current row and in column $col = ROW[row] + 0$ is not attacked.

From (1):

$m := m+1$;
if $m > 3$ **then** *highrow* := *row* := $m+2$ **else** *lowrow* := *row* := $4-m$;

[As revised acc. to point 30, see final program].
comment (2) is satisfied by condition for use;

From (3) and (4):

col := *ROW*[*row*] := 0;

Cluster 2: decrease m.

Conditions for use: $m > 0$, with the queens placed in the previous rows, the current row has been exhaustively tried, a queen being now present in the highest numbered column, $col = ROW[row] + 0$. [Revised according to point 31: ... tried, no queen being now present in it.]

From (3) and (4):

FREE[*col*] := **true**; [As revised according to point 31, delete this statement]
$m := m-1$;
if $m > 3$ **then** *row* := *highrow* := $m+2$ **else** *row* := *lowrow* := $4-m$;
if $m = 3$ **then** *highrow* := 5;
[As revised according to point 30, see final program.]
col := *ROW*[*row*].

Cluster 3: initialize m.

Conditions for use: the queens placed in rows 4 and 5 form a possible situation, registered in *ROW*[4], *ROW*[5], and *FREE*. *col* and *row* need not be set because they are set in the special use of cluster 1 below.
$m := 0$;
Perform immediately cluster 1.

21

Let us try to write the program, using Fig 5 and the clusters of point 20 and previous pieces of program. See Fig. 6 [To save repetition, the program of Fig 6 has been revised to show also later modifications, as indicated]. We have dropped the tables of *lowrow*, *row*, and *highrow* suggested in point 17, see clusters 1 and 2. [The test for diagonal capture is done by a modified form of point 15; this is again revised in point 30. The logic also deviates slightly from Fig. 5, in that the action *clear record of column* has been moved to $Q1$, just following Q].

Fig. 6.

```
begin
  integer c4, c5, m, row, col, lowrow, highrow, k, n, p, q;
  integer array Count, ROW[1:8]; Boolean array FREE[1:8];
    n := 0;
  for col := 1 step 1 until 8 do
    begin FREE[col] := true; Count[col] := 0 end;
  select(17); writetext( * <
  printer or typewriter: * );
  select(if lyn = 39 then 8 else 16); comment See point 24;
  writetext( * <
  8 queens—28.2.72—Peter Naur
  * );
  comment See Fig. 4 and 5;
  for c4 := 1, 2, 3 do
  begin ROW[4] := c4; FREE[c4] := false;
    for c5 := c4+2 step 1 until 9-c4 do
    begin ROW[5] := c5; FREE[c5] := false;
      writecr; write( * -dddd *, c4, c5); comment See point 26;

        comment Cluster 3;
        m := 0;

    P:  comment Cluster 1, revised acc. to point 30;
        m := m+1;
        if m > 3 then
        begin lowrow := 1; highrow := m+1; row := m+2 end
          else
          begin lowrow := 5-m; highrow := 5; row := 4-m end;
        col := ROW[row] := 0;
```

```
Q:  ;comment Select next possible column;
    if col ≠ 0 then
Q1: FREE[col] := true;
try for column:
    if col = 8 then go to reject row;
    col := col+1;
    if ¬ FREE[col] then go to try for column;
    FREE[col] := false;
    ROW[row] := col;
    comment Partial elimination of symmetry, see point 34;
    if abs(col-4.5) < 1 then
    begin if abs(row-4.5) > 5-c4 then go to reject column
    end;
    comment Try for diagonal capture, point 15 modified and
      revised according to point 30;
    for k := lowrow step 1 until highrow do
    begin
      if abs(row-k) = abs(col-ROW[k]) then go to reject column
    end;
    comment success;
    if m < 6 then go to P;
begin comment Final check for symmetry, just before printing,
  see point 34;
  integer S, R, a;
  integer array COLUMN[1:8];
  integer procedure solno(qu); integer qu; comment For a given
    value of the global a, qu must give a row (or column) number
    for a queen in column (or row) a;
  begin integer sum; sum := 0;
    for a := 4, 5, 3, 2, 1, 6, 7, 8 do
      sum := 10×sum+qu;
    solno := sum
  end solno;
  S := solno(ROW[a]);
  R := solno(ROW[9-a]);
  if R < S ∨ 99999999-R < S then go to reject column;
  for a := 1 step 1 until 8 do COLUMN[ROW[a]] := a;
  R := solno(COLUMN[a]);
  if R < S ∨ 99999999-R < S then go to reject column;
  R := solno(COLUMN[9-a]);
  if R < S ∨ 99999999-R < S then go to reject column
end final check for symmetry;
```



```
    comment Output as in point 22;
    n := n+1; writetext(⊀ <
    Solution ⊁); writeinteger(⊀ -ddd⊁, n);
    for p := 1 step 1 until 8 do
    begin writecr;
      for q := 1 step 1 until 8 do
        writetext(if q = ROW[p] then ⊀<x⊁ else ⊀<.⊁)
      end;
  reject column:
    Count[m] := Count[m]+1; comment See point 23;
    go to Q1;
  reject row: ;

    comment Cluster 2, revised acc. to point 30 and 31;
    m:=m-1;
    if m>3 then begin highrow := m+1; row := m+2 end
      else begin lowrow := 5-m; row := 4-m end;
    col := ROW[row];

    if m > 0 then go to Q;
  FREE[c5] := true
 end c5;
 FREE[c4] := true
end c4;
writetext(⊀ <
Count of rejection depth
⊁); comment See point 23;
for k := 1 step 1 until 6 do
  writeinteger(⊀ -ddddd⊁, Count[k])
end
```

22

Form of results: if we are to check them they must show the chessboard directly. Proposal in the form of example:

Solution 4

```
. . . x . . . .
x . . . . . . .
. . x . . . . .
```

etc.

That looks good, we take that.

23

To compare with the guesses in point 19 it would be nice to have some statistics on the search depth. So let us count how many times rejection is done for each value of m.

24

Output medium: it is nice to be able to use either typewriter or printer. We make this a manual choice.

25

Within the middle part of the program, from Q to *reject row*, m is >0. This is easy to prove: this section is entered the first time at P and nowhere else, with $m=1$. Decrease of m is done only after *reject row* and return jump is made only for $m>0$. This justifies that cluster 2 is used.

26

This type of program is unpleasant when first tried, because perhaps no output comes out because no solutions exist. To indicate that some progress is made let us therefore get something out on the way, $c4$ and $c5$ each time they are changed (12 times in all) would be nice.

27

I have been rather careless about the description of the state of the current row, not keeping $ROW[row]$ and $FREE[col]$ completely synchronized.

28

Program is punched and run once. Three mistakes, two uses of comments after label, one typing mistake.

29

After correction the program runs and produces a lot of solutions. Most of them are incorrect, however, because two or more rows can have $col=8$ for some reason. Point 27 above is relevant, no doubt! There seem to be two possible mistakes: 1) $FREE$ is not set correct when $col=8$, or 2) the selection of the next possible column fails. Reason 2 seems the more likely.

30

While hunting for the mistake by means of step-by-step execution at the desk, I realize that the test for diagonal capture can be still further simplified, by utilizing that the current row is always equal to *highrow* or *lowrow*. Just by a redefinition of these, to cover only those rows which

should enter into the diagonal capture test, the trick to avoid the attack on yourself becomes unnecessary.

31

The mistake: wrong use of cluster 2: condition not met, there is not a queen placed in $col=ROW[row]$!

32

Looking at the results of the third run and finding a lot of symmetric solutions, here is an approach how to eliminate them: the elimination will be done just before printing. We will have to look at the columns 4 and 5 and find out whether the pair of queens sitting there form a pattern which has already been produced before on rows 4 and 5, in our systematic generation procedure. If so, a symmetric solution with the one we have must already before have been printed. To detect this we must do the following steps: 1) Find $r4$ and $r5$, the row numbers of the queens in columns 4 and 5.

2) Remembering the ordering of the $c4$, $c5$ pairs (see Fig. 3): $c4$ goes 1, 2, 3; $c5$ goes $c4+2 \ldots 9-c4$, and further remembering the possibilities of reflexion symmetry, which means that it is as good to step down from 8 as it is to step up from 1, we can identify with $c4$ that one of $r4$ and $r5$, which has the largest difference, absolute, from the middle, 4.5.

33

The above approach to the symmetry problem cannot solve the problem completely, since in general it is not enough to look only at the rows and columns 4 and 5.

34

New attempt at a solution of the symmetry problem: In order to eliminate all cases of symmetry it is necessary to find a unique identification of each group of symmetric solutions. This is created as follows: First define the solution number of a solution held in one particular orientation to be the integer having the digits given as the values of the elements of ROW taken in the order in which they are released during the search: 4, 5, 3, 2, 1, 6, 7, 8. An octal number system would do, but decimal will do as well. Thus, for example the first part of the search for solutions, with $c4=1$ and $c5=4$, will look for solutions having their solution number with the leading digits 14. Now, for any given solution, rotate and reflect the solution into all of its 8 possible orientations and in each case determine the solution number. Now define the identification of this group of symmetric solutions to be the smallest of these 8 solution numbers.— The order in which the column numbers are selected will ensure that the

solutions are found in order of increasing identification number. However, as so far defined, the program will insert into this sequence of solutions, symmetric solutions of solutions already found before (ghosts). In order to determine whether a solution at hand is a ghost proceed as follows: 1) Find the solution number of the solution as it has been found, S. 2) Find the remaining 7 solution numbers, corresponding to the possible alternative orientations, R_i, $i=1 \ldots 7$. 3) If any R_i is $<S$ then the solution is a ghost.

There are two reasons for wishing to determine symmetry: to avoid them in the output, and to speed up the search by avoiding branches of the search tree where it is certain that all solutions are ghosts. The above test on final solutions serves the first purpose. To serve the second purpose we have to test on the relation between S and the R_i before a final solution is at hand. The choice of *c4* and *c5* is really a start on this, involving, however, only some of the R_i. As an example of how to continue in this direction, we might just test for the first digit of R_i. As soon as just one R_i has a first digit smaller than *c4* we know that we have a ghost. To do this we have to look into the queens placed in columns 4 and 5, thus: insert just before or just after the "Try for diagonal capture" the statements:

if *col* = 4 **or** *col* = 5 **then**
begin if *abs*(*row*−4.5) > 5−*c4* **then go to** *reject column* **end**;

(Whether this should be put before or after *Try for diagonal capture* must depend on whether for the work involved (average) it leads to a quicker rejection than does *Try for diagonal capture*).

It must be clear, however, that the above further step along the direction of eliminating symmetry during the search is only a help to eliminate some symmetric cases, and that the final elimination can only be made in the final set of R_i, just before printing of a solution.

35

Run time for the third run: 70 seconds. There were 27 solutions produced. Inspection eliminates the symmetries so 12 different solutions are left. Result of the count of rejection depth: 32, 96, 246, 332, 273, 83.

36

Plan for further work—summary of above points.

1) Point 30 should be done, in program and in the description of point 20.

2) Point 31 should be corrected in point 20.

3) The elimination of symmetric solutions according to point 34 should be done. This involves two changes in the program, first as shown in point 34, then the final elimination just before printing of a final solution.

4) The counts of rejection depth should be analysed and compared with the run time and the guesses in point 19.

37

With the changes 1, 2, and 3 of 36 the program produces 12 solutions, and runs in 50 seconds. Rejection depth counts: 33, 100, 247, 264, 208, 62.

38

Execution time (the present final point is a brief summary of more detailed work). Experiments give the following results related to the execution time of the final program:

Output of 12 solutions	12 sec.
Final elimination of symmetric solutions	12 -
Search for solutions	26 -

An analysis of the search part of the program, using the execution times of the algorithmic constituents given in the Manual of Gier Algol 4 (1967) and the rejection depth counts of point 37, gives a total search time of 27.6 seconds. In view of the known uncertainty of the analysis, this agrees perfectly with the observed time of 26 sec. A comparison with the estimates of point 19 comes out as follows:

	Point 19	Program
Number of queen pairs to check	720	4120
Total check time	1.0 sec.	8.3 sec.
No. of positions generated	1440	1378
Total time of generation	6.1 sec.	19.3 sec.

This comparison shows that the estimate of point 19 was unnecessarily crude. In fact, even on the same basic estimate, that on the average three more queens, beyond the two first, are required during the search, far more realistic estimates might have been made, with very little extra trouble. Thus the number of pairs to check should have been estimated: 12 start positions × 6 positions of third queen × (2 pairs + 5 positions of fourth queen × (3 pairs + 4 positions of fifth queen × 4 pairs)) = 7000 pairs. Similarly, the generation of positions could have been estimated more realistically.

3. Discussion.

After completion of the above developments I returned to the solution of Wirth [6]. In order to get the most concrete basis for comparison I tried to run his solution. In this way it became clear that solution given in section 6 of the paper, with its 9 lines, cannot fairly be compared with the solution given here. In fact, if expressed in terms of the proper details and declarations it expands to 40 lines. Also, the process of rewriting turned out to be not quite trivial. Thus the form of the procedure *regress* given by Wirth contains the following passage: **begin** *reconsiderpriorcolumn*; **if** $\neg$*regressoutoffirstcol* **then**. If expanded directly according to the directions given the result is: **begin** $j := j-1$; $i := x[j]$; **if** $j \geq 1$ **then**. This form causes the program to terminate with an improper reference to $x[0]$, however. To avoid this, the expansion should be changed to read: **begin** $j := j-1$; **if** $j \geq 1$ **then begin** $i := x[j]$; ...

The expanded version of Wirth's program produced 92 solutions, including all symmetric variants. With the output statements removed it required 83 seconds of Gier time.

Comparing with the approach of Wirth, the present development process may be characterized more generally as follows:

1. Symmetric solutions were eliminated. In view of the fact that at the start I did not have any ready idea for solving the symmetry problem, this added to the realism of the attack. The elimination of symmetry turned out to be a foreign element in the program, appearing in three different places. Systematically this is unsatisfactory, but it was accomplished without any special difficulty, because of the clarity of the description given in point 20.

2. The open pursuit of alternatives, many of which have been rejected, see points 2, 8, 9, and 11.

3. Decisions were made in large steps, see points 12 and 36. This is in contrast with the approach of stepwise refinement, and in small scale corresponds to the cycles of software project development described by Fraser in [5, p. 19].

4. Timing and efficiency considerations were explicit and quantitative, see points 1, 3, 12, 19, 35, and 38. Performance output, point 23, also belongs here.

5. The possibility of logical errors was taken into account realistically, see point 26.

4. Conclusion.

The primary aim of the present study has been to contribute some genuine empirical observations related to the program development process. For this reason the resulting program is to be taken only as an incidental result of the work. In the author's opinion this program is by no means ideal. In particular more of the actions related to the selection of the next current column should have been expressed in terms of action clusters. Also a proof of the correctness of the program is missing, although the most important elements of such a proof are available as the invariant assertions (1) to (4) of point 20.

What may perhaps be inferred from the observations is:

1. At least some program development does not, and can hardly be made to, proceed as the top-down process advocated by Dijkstra and Wirth. Rather, a problem-solving type of process is taking place. In this process high-level program descriptions certainly appear, but only after important details have been explored at a lower level.

2. The development of low-level action clusters and the associated invariant assertions appears as a usable tool, but only after a less formal exploration of the solution possibilities has taken place.

3. It is difficult to avoid the impression that program development styles have to allow for personality factors. More observations are required.

REFERENCES

1. E. W. Dijkstra, *Notes on structured programming*, Technical University Eindhoven, 1970.
2. R. Hyman and B. Anderson, *Solving Problems*, International Science and Technology, (Sept. 1965), 36–41.
3. P. Naur, *A manual of Gier Algol 4*, Regnecentralen, Copenhagen, 1967.
4. P. Naur, *Programming by action clusters*, BIT 9 (1969), 250–258.
5. *Software Engineering*, ed. P. Naur and B. Randell, Nato Science Committee, 1969.
6. N. Wirth, *Program Development by Stepwise Refinement*, Comm. ACM 14 (April 1971), 221–227.

DATALOGISK INSTITUT
UNIVERSITY OF COPENHAGEN
COPENHAGEN, DENMARK