Algorithms Supplement

Previously published algorithms

The following algorithms have recently appeared in the Algorithms Sections of the specified journals.

(*a*) **Communications of the ACM** (October–December 1969)

355 AN ALGORITHM FOR GENERATING ISING CONFIGURATIONS

Generates n-sequences $(S_1, \ldots, S_n)$ *of zeros and ones where*

$$x = \sum_{i=1}^{n} S_i$$

and

$$t = \sum_{i=1}^{n-1} |S_{i+1} - S_i|$$

are given.

356 A PRIME NUMBER GENERATOR USING THE TREESORT PRINCIPLE

Finds the first $m \geqslant 4$ *elements of the infinite sequence* 2, 3, 5, 7, 11, . . . *of prime numbers using a method of distinguishing primes from composite numbers similar to that of B. A. Chartres.*

357 AN EFFICIENT PRIME NUMBER GENERATOR

Finds the next m primes.

358 SINGULAR VALUE DECOMPOSITION OF A COMPLEX MATRIX

Finds the singular values $\sigma_1 \geqslant \sigma_2 \geqslant \ldots \geqslant \sigma_N$ *of the complex* $M \times N$ *matrix* $(M \geqslant N)$ *given in the first N columns of an array A. Also finds the first NU columns of an* $M \times M$ *unitary matrix U and the first NV columns of an* $N \times N$ *unitary matrix V such that* $||A - U\Sigma V^*||$ *is negligible relative to* $||A||$, *where* $\Sigma = diag(\sigma_i)$. *Can also be used to solve a homogeneous system of linear equations.*

359 FACTORIAL ANALYSIS OF VARIANCE

Transforms a vector y, observed in a balanced complete $t_1 \times t_2 \times \ldots \times t_n$ *factorial experiment, into an interaction vector z whose elements include mean and main effects.*

360 SHORTEST-PATH FOREST WITH TOPOLOGICAL ORDERING

Given a subset (called roots) of the nodes, numbered from 1 *to n, spanned by a directed graph composed of arcs of known length, finds for each node in the network the shortest path connecting it to its closest root node.*

361 PERMANENT FUNCTION OF A SQUARE MATRIX

Calculates the permanent function of an $n \times n$ *matrix A,*

$$per\ (A) = \sum_{r=0}^{n-1} (-1)^r \sum_{x \in T_{n-r}} \prod_{i=1}^{r} x_i$$

where $T_j, j = n, n-1, \ldots, 1$, *is the set of vectors* $\boldsymbol{x} = (x_i)$, $i = 1, 2, \ldots, n$, *which are obtained by adding j columns of A together in all* $\binom{n}{j}$ *possible ways.*

362 GENERATION OF RANDOM PERMUTATIONS

Produces a permutation on the integers 1, 2, . . ., *n, each of the n! permutations being given by one value of r between* 1 *and n! inclusive.*

363 COMPLEX ERROR FUNCTION

Evaluates the real and imaginary part of the function $w(z) = exp\ (-z^2)\ erfc\ (-iz)$ *for arguments* $z = x + iy$ *in the first quadrant of the complex plane.*

364 COLORING POLYGONAL REGIONS

Takes current regions and draws them in a two-dimensional array.

365 COMPLEX ROOT FINDING

Uses the downhill iterative method to determine, within a certain region, a root of a complex transcendental equation $f(z) = 0$ *with the only restriction that* $w = f(z)$ *must be analytic in the region considered.*

366 REGRESSION USING CERTAIN DIRECT PRODUCT MATRICES

A vector of observations is replaced by regression coefficients obtained by the matrix product C^T. *vector where* C^T, *the transforming matrix, is the direct product of the transposes of selected matrices.*

367 ANALYSIS OF VARIANCE FOR BALANCED EXPERIMENTS

Provides analyses of variance, covariance, and regression for data collected according to a wide variety of experimental designs.

(*b*) **BIT** (July 1969)

24 ALGORITHMS FOR THE INTERPRETATION AND EVALUATION OF A FUNCTION DEFINED BY CHARACTER STRINGS

Provides the possibility of supplying an arithmetic function to a computer program at execution time without previous compilation.

(*c*) **Applied Statistics** (March 1970)

AS26 RANKING AN ARRAY OF NUMBERS

Given a real array $v[1:n]$ *the procedure constructs another real array* $b[1:n]$ *where* $b(i)$ *is the rank of* $v(i)$.

AS27 THE INTEGRAL OF STUDENT'S t-DISTRIBUTION

Computes the area from t to + ∞ for Student's t-distribution using Gentleman and Jenkins' method.

AS28 TRANSPOSING MULTIWAY STRUCTURES

Overwrites real array X[1: n] with a re-ordered set defined by X[i] → X[j] where the inverse transformation is given in a subsidiary procedure map. Examples of map are given (i) *when X is a symmetric matrix stored as upper triangular by rows and is to be re-ordered so that it is stored as lower triangular by rows and* (ii) *when X is an n-way array which is to have its factor order permuted.*

The following papers, containing useful algorithms, have recently appeared in the specified journals.

(*a*) **Numerische Mathematik**

BALANCING A MATRIX FOR CALCULATION OF EIGENVALUES AND EIGENVECTORS (Band 13, Heft 4, pp. 293–304)

NUMERICAL CALCULATION OF ELLIPTIC INTEGRALS AND ELLIPTIC FUNCTIONS. III. (Band 13, Heft 4, pp. 305–315)

(*b*) **International Journal for Numerical Methods in Engineering**

A FRONTAL SOLUTION PROGRAM FOR FINITE ELEMENT ANALYSIS (Vol. 2, No. 1, pp. 5–32)

New algorithms

Algorithm 48

PROCEDURE FOR THE EVALUATION OF AN INTEGRAL OCCURRING IN THE MEAN SQUARE RESPONSE ANALYSIS OF LINEAR SYSTEMS

P. G. Littlewood
Dept. of Applied Mathematics and Computing Science
University of Sheffield

Author's note:

The procedure *intlap* evaluates integrals of the type

$$I = \int_0^\infty x^2(t)dt = \frac{1}{2\pi i}\int_{-i\infty}^{i\infty} X(s) \,.\, X(-s)ds$$

where

$$X(s) = \int_0^\infty x(t) \exp(-st)dt$$

and is a rational function of s with the degree of the numerator less than that of the denominator, i.e.

$$X(s) = c(s)/d(s)$$

where

$$c(s) = \sum_{k=0}^{n-1} c_k s^k$$

and

$$d(s) = \sum_{k=0}^{n} d_k s^k,$$

and $X(s)$ has poles in the left half-plane only.

In engineering investigations, integrals of the second type occur very frequently, often where n is very large. Fuller (1967) has given a short historical account of the analytical methods used to evaluate integrals of this type, most of which depend upon contour integration or are based on integration by parts. An alternative method, using autocorrelation functions, is given in Huntley (1969).

When $X(s)$ is a rational function of s, with all its poles in the left half-plane, then I can be expressed as an algebraic function of the coefficients $c_0, c_1, \ldots, c_{n-1}, d_0, d_1, \ldots, d_n$, and tables for I have been published (Newton, Gould and Kaiser, 1964) for $n \leqslant 10$. Unfortunately these tables become extremely laborious to use for $n > 7$. Fuller quotes a general expression for I, involving the ratio of two determinants and the method used, in this algorithm, for the evaluation of I, is closely related to this expression.

It can be shown that $I = (-1)^{n-1}x_n/(2d_n)$ where $x_1, \ldots, x_n$ satisfy the system of linear equations

$$\sum_{i=0}^{m} d_{m-i}x_{i+1} = g_{m/2} \qquad 0 \leqslant m \leqslant n-1$$

$$\sum_{i=m-n}^{n-1} d_{m-i}x_{i+1} = g_{m/2} \qquad n \leqslant m \leqslant 2n-2$$

where m takes even values only, and where the g_i are the coefficients of s^{2i} in the expansion of $c(s) \,.\, c(-s)$, i.e.

$$c(s) \,.\, c(-s) = g_0 + g_1 s^2 + g_2 s^4 + \ldots + g_{n-1}s^{2(n-1)}.$$

(If the system of linear equations were solved for x_n, by Cramer's Rule, then the resulting expression for I would be identical to that given by Fuller.)

The algorithm solves this system of equations by Gaussian Elimination, considering only the non-zero entries in the matrix of coefficients. In so doing it will be approximately four times quicker in execution than a method which evaluates, using standard routines, the determinants in the expression quoted by Fuller.

The algorithm has been extensively checked, for a variety of problems of different orders, and, in particular, with a tenth order system solved by the method described in Huntley (1969), in which complete agreement was obtained to nine significant figures.

References

FULLER, A. T. (1967). The Replacement of Saturation Constraints by Energy Constraints in Control Optimization Theory, *Int. J. Control*, Vol. 6, No. 3, pp. 201–227.

HUNTLEY, E. (1969). The serial/matrix technique applied to the analysis of linear systems with stationary random inputs, *Int. J. Control*, Vol. 10, No. 1, pp. 13–27.

NEWTON, G. C., GOULD, L. A., and KAISER, J. F. (1964). *Analytical Design of Linear Feedbak Controls*, J. Wiley.

```
procedure intlap(n, c, d, result);
value n, c, d; integer n; real result; array c, d;
comment intlap evaluates the integral of [x(t)] ↑ 2, with respect
to t, in the range 0 to infinity, in the cases where the Laplace
transform of x(t) can be expressed as a rational function of the
Laplace transform variable s.
i.e. x(s) = c(s) / d(s)
where c(s) = c(0) + c(1)s + . . . + c(n − 1)s ↑ (n − 1)
and d(s) = d(0) + d(1)s + . . . + d(n − 1)s ↑ (n − 1) +
d(n)s ↑ n.
Equivalently the procedure evaluates 1 / (2 × pi × i) times the
integral of x(s) × x(−s) with respect to s in the range
−i × infinity to +i × infinity, where i = sqrt(−1) and x has
the form defined above. Input to the procedure is given by
n: the order of the polynomial d(s)
c: the coefficients of the polynomial c(s) arranged as a vector
d: the coefficients of the polynomial d(s) arranged as a vector.
The result of the integral evaluation is obtained via the para-
meter result;
if n = 1 then result := 0·5 × c[0] ↑ 2 / (d[0] × d[1]) else
  begin integer i, j, k, l, m; real acc;
  array g[0 : n − 1], a[1 : n, 1 : n];
  j := −1;
  for i := 0 step 1 until n − 1 do
    begin
    comment this block evaluates the g's;
```

```
    l := j := −j; k := 0;
    acc := 0;
L1: if i + k = n − 1 or i − k = 0 then goto L2 else
      begin
      k := k + 1; l := −l;
      acc := acc + l × c[i − k] × c[i + k]
      end;
    goto L1;
L2: g[i] := 2 × acc + j × c[i] × c[i]
    end;
  for i := 1 step 1 until n do
    begin
    comment this block evaluates the coefficient matrix;
    m := i ÷ 2; k := if i / 2 − m < 0·1 then − 1 else − 2;
    l := (n − i + 1) ÷ 2 + i;
    for j := m + 1 step 1 until l do
      begin
      k := k + 2; a[j, i] := d[k]
      end
    end;
  for k := 1 step 1 until n − 1 do
    begin
    comment this block performs the Gaussian elimination;
    m := (n − k + 1) ÷ 2 + k;
    for i := k + 1 step 1 until m do
      begin
      a[i, k] := a[i, k] / a[k, k];
      if k = 1 then goto L3;
      l := if 2 × k − 1 < n then 2 × k − 1 else n;
      for j := k + 1 step 1 until l do
      a[i, j] := a[i, j] − a[i, k] × a[k, j];
    L3: g[i − 1] := g[i − 1] − a[i, k] × g[k − 1]
      end
    end;
  result := 0·5 × g[n − 1] / (d[n] × a[n, n]);
  if n / 2 − n ÷ 2 < 0·1 then result := − result
end intlap;
```

Algorithm 49

INDEXING SUBARRAYS IN MULTIDIMENSIONAL ARRAYS

K. W. Smillie
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, Canada

Author's note:

We are given an N-dimensional rectangular array, the elements of which are stored in row major order as a vector Y. The dimensions of the array are stored as an N-component vector R. We are required to select sequentially the elements of any specified rectangular subarray without first extracting the subarray from the array.

Let V be an N-component vector of non-negative components, such that the positive components give the fixed coordinates of the subarray in the array. Let the scalar B be the sequence number of any element in the subarray. For example, for a $5 \times 3 \times 4$ array, $R = (5, 3, 4)$, and Y has 60 components. The vector $V = (2, 1, 0)$ specifies the elements on the straight line defined by fixing the first and second coordinates at 2 and 1, respectively. The elements on this line may be selected by letting B take the values 1, 2, 3 and 4.

An algorithm to select the Bth element of a subarray may be described as follows: (1) Construct a vector W which is the representation of $B - 1$ in a mixed-base number system defined by those components of R corresponding to zero components in V. (2) Expand W to an R-component vector X with zeros in those locations corresponding to non-zero components in V. (3) Replace the zero components of X produced in (2) by 1 less than the corresponding components in R. Call the result Z. (4) Convert Z to a scalar I which is 1 greater than the decimal equivalent of Z considered as a base-R number. (5) Select the Ith component of Y as the Bth element in the subarray.

The function *SUBARRAY* for this algorithm is written in APL\360. B and V are arguments, E is the required element of the subarray, and R and Y are global variables defined outside the function. Each of the five statements of the function *SUBARRAY* 0 given as

```
    ∇ E←V SUBARRAY0 B; I; W; X; Z
[1]     W←( (V = 0)/R)⊤B − 1
[2]     X←(V = 0)\W
[3]     Z←X + (V ≠ 0) × V − 1
[4]     I←1 + R⊥Z
[5]     E←Y[I]
    ∇
```

performs the operations described in the correspondingly numbered step in the algorithm. The single statement in *SUBARRAY* combines the five statements of *SUBARRAY* 0 into a single statement.

Reference

GOWER, J. C. (1968). Simulating multidimensional arrays in one dimension, *Applied Statistics*, Vol. XVII, No. 2, pp. 180–185.

```
    ∇ E←V SUBARRAY B
[1]   E←Y[1 + R⊥((V ≠ 0) × V − 1) +
      (V = 0)\((V = 0)/R)⊤B − 1]
    ∇
```

Algorithm 50

HOW TO PROGRAM A COMPUTER TO PLAY LEGAL CHESS

A. G. Bell
Atlas Computer Laboratory
Chilton
Didcot

Author's note:

Many of the programs which have been written to play chess (Good, 1967) include restrictions. This paper describes how to produce all legal moves for any chess position including queening, castling and *en passant* captures. To test and demonstrate the techniques described a program was written in ALGOL to solve any two move mate problem. The program is given and the tables required to drive it are given with some optional procedures in full in Appendices 1–4.

The tables

The most important feature of the program is that it requires six tables (arrays) to drive it. To keep the program small it is best if these tables are read in rather than computed, and for this reason they are given in full in Appendix 1. Moreover it is simpler to understand and check how they work.

The six tables fall naturally into three groups, these are the knight and king, the bishop and rook, and the white pawn and the black pawn (2.3). (The moves of the queen are obtained by treating her as a rook and then a bishop.)

The knight and king tables

Appendix 1.1*a* is the table for the knight. It consists of 64 rows (each row corresponds to a square on the board as given in **Fig. 1**(*a*)), the numbers in the left-hand column being only for reference and not part of the table.

Consider row 1. This is used if a knight is in square 1. The last number of the row is the total number of squares to which a knight can move (in this case only 2), and the

preceding two elements are the numbers of the squares (18 and 11). The zeros in the table are not used.

The table of moves for a king (Appendix 1.1*b*) is constructed in exactly the same way, so the procedure '*knight or king*' in the program only requires the number of the square occupied by a knight or king, coupled with the appropriate table to read off the squares the piece can move to. The moves are stored sequentially in a list called '*moves*'. Note that two checks are made before accepting a move (these checks apply to all the men when listing their moves). The first is that no man is allowed to occupy a square which contains a friendly man, and secondly, that if the man can move to the square containing the opponent's king, then the listing of further moves is terminated (by exiting via the label '*cut off*'). This latter device checks the legality of the opponent's previous move. Castling the king is dealt with in the section '*Castling*'.

57	58	59	60	61	62	63	64
49	50	51	52	53	54	55	56
41	42	43	44	45	46	47	48
33	34	35	36	37	38	39	40
25	26	27	28	29	30	31	32
17	18	19	20	21	22	23	24
9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8

BLACK

N

S

WHITE

(*a*) Board notation

WB					WN		
		WB					
					BP		
WR			BN	BQ	BK	WP	
			BP		BP		WP
	WN		BP		WP		
	WK		WP		WQ		

(*b*) Example of actual position

3					2		
		3					
4						1	
							1
	2				1		
	6		1		5		

(*c*) 'whitemen'

					1		
			2	5	6		
			1		1		
			1				

(*d*) 'blackmen'

11	10	12	14	18	22	32	33	39	51	57	62

(*e*) Number and positions of white's eleven men—'whitepiece'

7	20	28	30	36	37	38	46

(*f*) Number and positions of black's seven men—'blackpiece'

Fig. 1. Representation of white and black men on the board. White plays from south to north

The rook and bishop tables

Appendix 1.2*a* is the table used to generate the rook's moves. The first row of numbers is the increment to move the rook from its initial square in each of its four possible directions, i.e. east (+1), west (−1), north (+8) and south (−8). Again consider row 1 which is used if a rook is in square 1. The first number is the terminal square on the board (see Fig. 1(*a*)) if it moves in an easterly direction (i.e. 8), the second if it moves west (1), the third if it moves north (57) and the fourth if it moves south (1). Attempts to move west and south from square 1 will be skipped.

The greatest number of moves a rook can have is 14, all of which would be produced only if the rank and file of the rook were empty, or if only the terminal squares contained enemy pieces. However, as in the '*knight or king*' procedure, each square generated must be tested for the presence of a friendly man. If he is present, then the move to that square is ignored and the next increment (i.e. direction) is selected:

if *my man in* [*i*] ≠ 0 **then goto** *new direction*;

If the square does not contain a friendly man, then the move is added to the list '*moves*' and a new test is now made for the square being occupied by a hostile man:

if *opponents* [*i*] ≠ 0

If he is absent the next square in the current direction is generated. Otherwise, the procedure will exit if that man is the opponent's king (like the '*knight or king move*') or else it will select another direction. When it has exhausted all four directions, it terminates.

The bishop table (Appendix 1.2*b*) operates in exactly the same way. The four directions this time are north-east (+9), south-west (−9), north-west (+7) and south-east (−7).

The white and black pawn tables

Unlike the pieces, the colour of pawns is important. In the program, white pawns move up the board and black pawns move down, hence two tables. The pawn is the most difficult man of all for which to generate moves because it has five distinct moves.

Consider the white pawn:

(1) from its starting position it can move either one or two empty squares up the board;
(2) from then on it advances 1 empty square at a time;
(3) captures opponents in the immediate north-east or north-west square;
(4) becomes a queen, rook, bishop or knight on reaching the eighth rank;
(5) captures *en passant*.

The black pawn differs only in that it must always move down the board.

The tables deal only with moves of the type (1), (2) and (3), queening (4) is dealt with by the program (see section—Making a move) and *en passant* captures by the section '*en passant capture*'.

Appendix 1.3*a* is the table used to generate the white pawn's possible moves. Pawns can only exist in the squares 9 to 56 and the rows are numbered accordingly. The first column is the pawn moving forward one square and if this square is empty the move is recorded. Then, and only then, the program will try to advance the pawn two squares (the number of this square is in the second column). If that square is also empty then the move is also recorded. Note that once a pawn has moved, the double move derived from the table is always suppressed because the square is apparently occupied (by itself!).

The third and fourth columns give the squares which may be entered provided an opponent is in them (i.e. captures). Square 65 of the opponent is always empty (a 'no-man's land'), catering for edge effects.

The black pawn's table (Appendix 1.3*b*) operates in exactly the same way. Once again, the procedure '*white or black pawn move*' only requires to know the square a pawn occupies and the table appropriate to the pawn's colour. The usual test for taking the opponent's king is made.

Listing the moves

The procedure '*list moves*' uses the three procedures described in the previous section and works for either black or white men. It must however distinguish the different men, and therefore each man is described by a different integer. Thus, king (K) = 6, queen (Q) = 5, rook (R) = 4, bishop (B) = 3, knight (N) = 2 and pawn (P) = 1.

Consider Fig. 1(*b*) in which the position is a two move mate for white. Figs. 1(*c*) and 1(*d*) are the required representation of the position stored in the arrays 'white men' and 'black men'.

'list moves' has five parameters:

(1) '*piece*'—an integer array (dimension 0 : 16) describing:
  (*a*) how many white or black men are on the board (in *piece* [0]), and
  (*b*) where each of them is (see Figs. 1(*e*) and 1(*f*)). This array saves time scanning the board for men but loses time when moves are actually made;
(2) '*mymanin*'—a representation of the board containing the pieces of the side whose moves we wish to list (see Fig. 1(*c*));
(3) '*opponents*'—similar to (2) but containing only the opponent's pieces. This is used to detect captures and therefore must be separate from '*mymanin*' (see Fig. 1(*d*));
(4) '*startat*'—the generated moves stored in a large array called '*moves*' begin at '*moves*[*startat*]';
(5) '*cutoff*'—a label used when the opponent's king can be taken, i.e. his last move was illegal.

On entry, the information generated will be stored sequentially in the array '*moves*' starting from element '*startat*'. For the example given in Figs. 1(*b–f*), a loop is set up to scan the 11 men

for *pointer* := *piece* [0] **step** −1 **until** 1 **do**

and '*square*' successively becomes the number of the square each white man occupies. If the square is empty, the man has been captured. However if there is a man there, then he is first identified (by storing 3 items of information in the array '*moves*') before generating his moves. The information is:

(1) '*pointer*'—his position in the array '*piece*';
(2) '*mymanin*[*square*]'—his value (pawn = 1, knight = 2, etc.);
(3) '*square*'—his position on the board;

and is used later when the moves are actually done.

The next instruction in the procedure is crucial. '*moveof*' is declared as a switch and the result of

'**goto** *move of* [*mymanin*[*square*]]'

is to call the appropriate procedure with the correct parameter. Note the colour of a pawn is checked to decide which table to use, and the moves of a queen are the result of a rook and then a bishop call.

In the example, the 11th man is considered first (the loop runs faster if written this way). He is in square 62 of the 'white men' board which contains the number 2, i.e. a knight. Therefore, '*moves*[*c*]' is 11, '*moves*[*c* + 1]' is 2 and '*moves*[*c* + 2]' is 62.

The program then jumps to label '*a knight*', obeys the call '*knight or king move* [*knight*]' and generates the moves for a knight located in square 62, i.e. the list 56,47,45,52. The program then jumps to label '*nextman*' which checks if any moves have been generated (the identification will be over-

	NO.	MAN	POSITION		BACK TO
knight	11	2	62	56, 47, 45, 52	−62
bishop	10	3	57	50, 43, 36	−57
bishop	9	3	51	60, 42, 58, 44, 37	−51
pawn	8	1	39	47, 46	−39
rook	7	4	33	34, 35, 36, 41, 49, 25, 17, 9, 1	−33
pawn	6	1	32	40	−32
knight	4	2	18	35, 28, 3, 1	−18
queen	3	5	14	15, 16, 13, 6, 23, 5, 21, 28, 7	−14
king	1	6	10	1, 2, 3, 9, 11, 17, 19	−10

Fig. 2. List of moves of white for position in Fig. 1

written if the man cannot move). If so, it stores '*—square*' in the moves list (i.e. −62 in the case of this knight) to indicate the termination of the man's moves, and then considers the next man in the list '*piece*'. It similarly continues until the list is exhausted. The resulting list of numbers in '*moves*' is given in **Fig. 2.** Note that two pawns (numbers 5 and 2) have no moves and are omitted.

After generating the moves of all the men, the position of the last entry is recorded in '*startat*'. This marker is used when the moves are actually being made. Illegal moves (the king cannot move to squares 11 or 19) will be detected when they are actually made and it is found that the opponent can then take the king. (See next section.)

Making a move

The procedure '*make move*' operates on the list generated by '*list moves*'. Its functions are:

(1) to alter the contents of the four arrays '*whitemen*', '*blackmen*', '*whitepiece*' and '*blackpiece*', depending on which side has the move (if the depth is odd then white, otherwise black).

To do this, it needs to know:

(*a*) the squares from and to which a man can move ('*pointer*' into the list '*moves*' gives this information. In the example, '*pointer*' equals 3 for the knight's first move, hence '*from*' is 62 and '*to*' is 56);

(*b*) the position in array '*mypiece*' of the man it is about to move (in the example the knight is in the 11th position). This information is in '*locate*[*depth*]'; and finally

(*c*) the initial value of the man in case it is a pawn which queens. This information is in '*q*[*depth*]'.

To move a man requires six operations labelled *A*1 to *A*6 in the program:

*A*1: the value of the man ('*q*[*depth*]') is stored in '*to*' (unless it is a pawn queening in which case a queen, rook, bishop and knight substitution is successively made);

*A*2: the square '*from*' is cleared;

*A*3: any possible capture on the man's previous move (when the square '*from*' was entered) is restored to the opponent's board;

*A*4: any possible capture in the new square is now recorded, '*c*[*depth*]' usually becomes zero but this does not matter;

*A*5: clear his board of any possible man in square '*to*';

*A*6: update the list of positions of men on the board. In the example the knight in the 11th position occupied square 62, this is now replaced by 56 ready to generate white's list of second moves.

(2) After making a move, the depth is increased and the opponent's replies are generated for the new board configuration. This tests if the move just made is legal, and if not, the call of '*listmoves*' will return to the label '*illegal move*' effectively ignoring the move just made.

(3) If the move is legal, the list of opponent's replies is ready to be made. The number of the first opponent and his value are initialised and the procedure terminates (the purpose of '*back to* [*depth*]', which records the initial square of the man just moved, is explained in the next section).

The effect of '*make move*' operating on the list '*moves*' is to cycle a man around the squares he can move to until a negative '*to*' square is reached. This means that the man has completed his moves. The man and board are, therefore, returned to their starting positions (the five operations labelled *B*1 to *B*5. N.B.—We know '*hismen*[*—to*]' must be zero, otherwise the operations are identical to *A*1–*A*6). The next man's moves are initialised and the procedure exits via label '*continue*' which checks that the list of moves has not been exhausted and usually results in an immediate call of the procedure '*make move*' again and the new man begins his moves.

Solving two move mates

In order to test the validity and speed of the procedures '*listmove*' and '*makemove*', the complete program uses them to solve any two move mate problem with white to move. The program works in the same way as a human solves the problem albeit rather moronically. That is, it makes white's opening moves, one after the other, until it finds the opening move which, regardless of what black does in his next two moves, allows white to check on his second move (to avoid stalemate) and to always capture the black king on his third move.

To speed up the solution it is necessary to reverse a move at times. For example, white makes an opening move (W1), and black makes a reply (B1) such that white cannot check

(*a*)

					BB	BR	BK
				BP		BP	
				WP	BP	WP	
				BP	BP	BP	
				BP		WP	
				BP		WP	WB
				WP		WP	WR
					WB	WR	WK

Bell's position—the simplest two move mate position known

(*b*)

11	6	7	8	13	15	16	23	24	31	45	47

'white piece'

11	21	29	37	38	39	46	53	55	62	63	64

'black piece'

Fig. 3. Position is two move mate inevitably. White has only one move, black has only one (legal) reply and white still only has one move

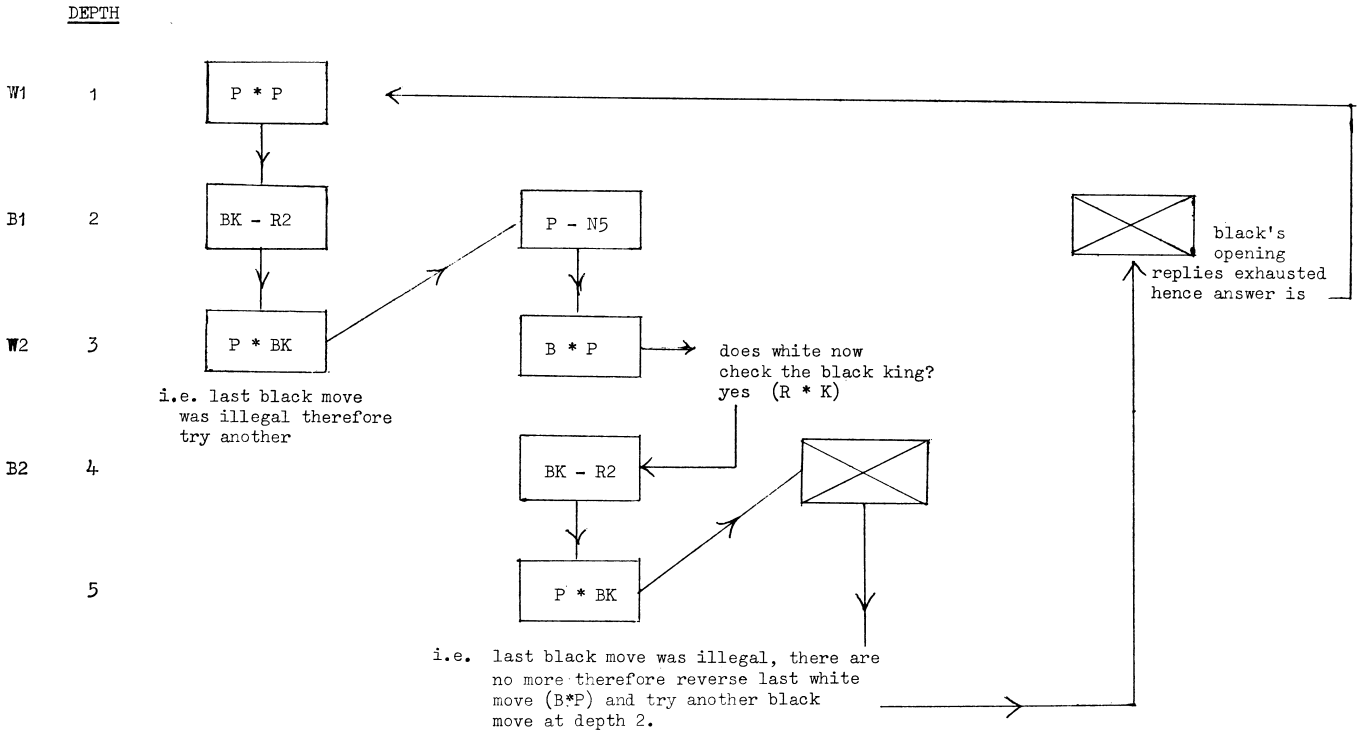

Fig. 4. Flow of program when solving problem given in Fig. 3

the black king with his second move. Now it is necessary to reverse the black reply (B1) before attempting another white opening move. This is done by calling procedure '*reverse move*', giving the present square that the black man occupies and his initial square. The initial square of all men, before beginning to make that man's moves, is always stored in '*back to*[*depth*]'. In order to show the flow of the solution, consider the position given in **Fig. 3**(*a*). There are 11 white men and 11 black men in the positions described by the arrays '*white piece*' and '*black piece*' (Fig. 3(*b*)).

When black's moves at depth 2 are exhausted, the solution has been obtained and the position achieved by the correct white opening move can be printed.

The flow of the program is given in **Fig. 4** and the actual contents of the array '*moves*', giving the moves generated at the five depths, is given in **Fig. 5.** When run on Atlas the program can solve a two move mate problem in about 45 seconds, this reduces to 5 seconds when written in basic code. It can also prove the uniqueness of the answer.

	DEPTH	NUMBER OF MAN	INITIAL VALUE	SQUARE	TO	ETC.
W1	1	9	1	31	38	−31
			P	*	P	
B1	2	11	6	64	56	−64
			K	—	R2	
					(illegal)	
W2	3	8	3	24	31	−24
			B	*	P	
B2	4	11	6	64	56	−64
			K	—	R2	
	5	11	1	47	56	listing of moves terminated
			P	*	K	

5 1 39 31 −39
P — N5

Fig. 5. Contents of array 'moves' on completion of problem given in Fig. 3. Compare with Fig. 4

Omissions

The program to solve two move mates will fail in two cases:

When
(1) castling; or
(2) *en passant* capture is involved.

These are omitted because they slow the program down and rarely occur in such problems. However, if required, they may be included as follows.

Castling

For any given position, the program requires four Boolean variables to be set to calculate whether white or black can castle on either the king or queen side. Thus

white king side castle := *white queen side castle* :=
black king side castle := *black queen side castle* := **true**;

would be used if the program is presented with a position in which all castlings are still valid (**Fig. 6**(*a*)). To simplify the problem, the positions of the king, the king rook and the queen rook in the array '*piece*' should be fixed for both sides (e.g. 1, 2 and 3 as given in Fig. 6(*b*) and 6(*c*)).

To castle white on the king side merely needs the following list of numbers inserted in the array '*moves*'

1 6 5 −7 2 4 8 6 −8 1 6 7 −5
K — KN1 R — KB1 uncastle

The procedure '*make move*' will then operate correctly.
To castle white on the queen side requires the list

1 6 5 −3 3 4 1 4 −1 1 6 3 −5
K — QB1 R — Q1 uncastle

To castle king side requires

1 6 61 −63 2 4 64 62 −64 1 6 63 −61
K — KN1 R — KB1 uncastle

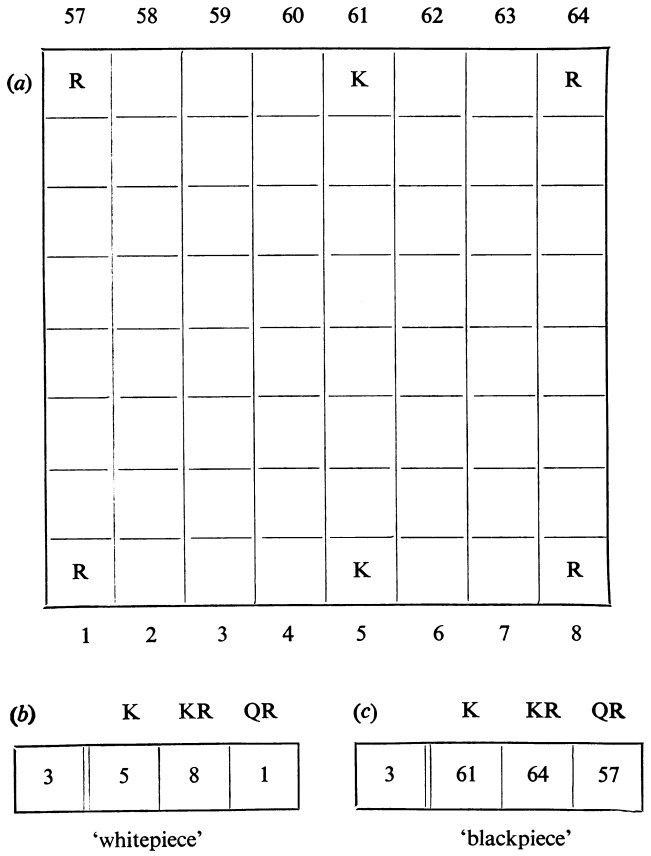

Fig. 6. Castling

and to castle black queen side requires

```
1  6  61  −59  3  4  57  60      −57  1  6  59  −61
   K   —  QB1     R   —  Q1           uncastle
```

The difficulty is whether the appropriate list can be legally added to the list '*moves*'. Three questions must be asked.

Q1: Have the king or rooks moved? (The four Boolean variables help to answer this one.)
Q2: Any pieces between king and rook?
Q3: Does the enemy control any of the three squares the king is in, must pass over or go to?

Appendix 2 is a piece of ALGOL text which asks these questions and, if successful, inserts the correct list. Points to note are:

(1) At any depth, men moved to obtain a position have been recorded in '*locate* [*depth*]'. So question 1 is answered by scanning the contents to see if king or relevant rook has been moved, plus the knowledge whether it was possible to castle (king or queen side) in the initial position;
(2) castling is the only move which is checked for legality *before* being placed in the list, because the moving of two men is involved and '*reverse move*' can only return one man to his initial position.

The piece of program has been written in full in order to show exactly the conditions being tested. It can be easily compressed. The procedure '*castle*' replaces the dummy procedure in the program.

En passant capture

To implement *en passant* capturing requires the use of the array '*en passant*' which records the position of the '*ghost*' pawn when a pawn moves forward two squares. The following four operations must be made and the associated code inserted in the program at the labels *ep*1, *ep*2, *ep*3 and *ep*4 (twice) respectively.

1. Detect a pawn has moved forward two squares and, if so, record position of ghost pawn:

```
ep1: en passant[depth] := if q[depth] = 1 and
     abs (backto[depth] − to) > 10
     then (backto[depth] + to) ÷ 2 else 0;
```

2. When listing pawn moves, check if a pawn can move to capture a possible ghost pawn, and if so, record the move:

```
ep2: if opponents [pawn[i]] ≠ 0 or
     pawn [i] = en passant[depth − 1]
     then
```

3. When making a pawn move, ask if it is an *en passant* capture, and, if so, apparently capture a man with a negative value and remove the correct pawn:

```
ep3: if q[depth] = 1 and to = en passant[depth − 1]
     then
       begin his men [to] := −1;
       his men [if to < 32 then to + 8 else to − 8] := 0
       end;
```

4. When reversing a move, ask if '*c*[*depth*]' is negative, i.e. the move was an *en passant* capture. If yes, then restore correct pawn. (N.B.—This is required in two places in the program.)

```
ep4: if c[depth] = −1 then
     his men [if from < 32 then from +8 else from −8] := 1
     else his men [from] := c[depth];
```

Input and output

It is recommended that the tables and the chess position be input as shown in Appendix 3. The procedure '*print position*' given in Appendix 4 is useful when debugging the program. Both use the Atlas I/0 statements (I.C.T., 1966) and are self-explanatory.

Summary and applications

This paper is mainly concerned with describing techniques to generate a list of all legal moves for any chess position. The problems of:

(*a*) how many times a position has occurred before with the same player to move; and
(*b*) how many moves have been made since the last capture or pawn move, are not solved. It can become very difficult if they are included in the look ahead, but it is not illegal to ignore them. Moreover, such situations are not likely to occur in a game.

It is hoped that the paper will be of use to people who wish to write a chess-playing program; of course the techniques (especially the contents of the tables) are not sacrosanct and should be adapted to the computer and language one wishes to use. The main points to appreciate are:

(1) The table drive.
(2) How to check the legality of a move, i.e. by actually making it and calling opponent's '*list moves*'. This avoids the problems of pins and moving into check with which some programs have wasted time and, if legal, one is immediately ready to execute the opponent's replies.
(3) How to queen, castle and capture *en passant*. No previous chess program has included all these features before, they are idiosyncracies of the game and do not fit easily into a table-driven program.
(4) The fact that previous positions are not remembered as such but information (hopefully a minimum) is

stacked from which the program restores a position when required. Again some chess programs actually stored a copy of each new position a player could reach. Thus, to solve a two move mate problem would require 64 (number of squares) × 30 (average number of moves) × 4 (plies) = 7680 locations not to mention the time wasted in storing them.

Having understood these points, which suffice for solving simple mate problems, extensions are required for other aspects of chess. For example, when playing a game further information must be produced by '*list moves*', such as number of squares controlled and their relative importance (control of centre squares is better than that of edge squares).

Also, because ALGOL usually runs 5–10 times slower than the equivalent basic code, the program should be coded in one of the lower languages of a particular machine. This is not as difficult as it seems because the tables are, on the whole, machine and language independent. Translation of this program into Atlas Basic Language took about 10 days.

The program was also run successfully on the C.D.C. 6600 ALGOL Compiler, requiring 1 day to punch the program in the dialect and alter the input/ouput statements.

Finally, and mainly as a demonstration, the ALGOL program was extended to actually play a game via the on-line console at the S.R.C. Atlas Laboratorv. Much of the effort went into producing an agreeable input/output system.

The strategy was simple, the program looked ahead 3 plies and would accept equal or better captures in that range with weighting on swops of the more powerful pieces (i.e. it will always swop a queen for a queen); if no swops or captures were present it attempted to control as many squares as possible. This program, although written in ALGOL, responded almost immediately to the moves of the opponent when entered via the on-line console.

The program played two games with John Scott's program on the ICL 1900 computer (Scott, 1969) and the result is given in **Fig. 7.** N.B.—Only two games are possible because neither program learns. The strategy of the ALGOL program was altered at move 4 in game 1 to take advantage of the position, i.e. I cheated.

The strategy of taking equal or better captures seems to work and results in dull but fairly equal games in these two cases. The Scott program responds in the order of 1 minute and is coded in basic 1900; it therefore appears that, as chess programs spend a great deal of time listing moves, the techniques described in this paper are reasonably efficient.

Papers on the problem of playing good chess on a computer are Greenblatt's chess program (Greenblatt, 1967); the playing techniques described by Arthur Samuel, especially Alpha-Beta pruning (Samuel, 1967) and the attempt by Barbara Huberman to analyse end games (Huberman, 1968).

Acknowledgements

The author would like to thank the staff of the Atlas Computer Laboratory for its aid in producing this paper.

GAME 1

	SCOTT(W)	v. ATLAS(B)
1.	P–Q4	N–QB3
2.	P–K4	P–K3
3.	P–Q5	B–QN5 ch.
4.	P–QB3	B–QB4
5.	P–QN4	N*P
6.	P*N	Q–B3
7.	P*B	Q*R
8.	B–Q3	Q*P
9.	N–KB3	P*P
10.	KP*P	Q*QP
11.	N–QB3	Q*P
12.	B–Q2	
13.		

GAME 2

	ATLAS(W)	v. SCOTT(B)
1.	P–K4	P–Q4
2.	P*P	Q*P
3.	P–Q4	N–KB3
4.	B–KB4	Q–K5 ch.
5.	B–K3	B–Q2
6.	P–Q5	Q*QP
7.	Q*Q	N*Q
8.	B–QB5	P–K4
9.	B*B	K*B
10.	B–Q3	N–KB5
11.	B–QB4	N*P ch.
12.	K–Q2	P–KR4
13.	N–QB3	R–KR3

Fig. 7. Game played by modified program

References

GOOD, I. J. (1967). A Five-year Plan for Automatic Chess, *Machine Intelligence* 2, Oliver and Boyd: Edinburgh.

GREENBLATT, R. D., EASTLAKE, D. E., and CROCKER, S. D. (1967). The Greenblatt Chess Program, AFIPS Conference Proceedings: Thomson.

HUBERMAN, BARBARA J. (1968). A Program to Play Chess End Games, Computer Science Department, Stanford University, Technical Report No. CS 106, August.

SAMUEL, A. L. (1967). Some Studies in Machine Learning using the Game of Checkers. II—Recent Progress, *IBM Journal*, November.

SCOTT, J. J. (1969). A chess-playing program, *Machine Intelligence* 4, Edinburgh University Press, pp. 255–265.

```
begin
integer i, j, k, l, m, n, depth, w1, b1, w2, b2;
integer array knight, king[1:576], bishop, rook[0:259],
  wpawn, bpawn[36:227], whitepiece, blackpiece[0:16],
  whitemen, blackmen[1:65], c, q, locate, numberofmoves,
  backto, en passant[1:5], moves[1:500];

procedure print position;
See Appendix 4: ;

procedure listmoves(piece, mymanin, opponents, startat,
  cutoff);
integer array piece, mymanin, opponents;
integer startat; label cutoff;
  begin
  switch moveof := apawn, aknight, abishop, arook, aqueen,
    aking;
  integer c, i, j, k, l, pointer, square;

  procedure castle;
  See Appendix 2: ;

  procedure knightorkingmove(knightorking);
  integer array knightorking;
    begin
    j := 9 × square;
    for i := j − knightorking[j] step 1 until j − 1 do
    if mymanin[knightorking[i]] = 0 then
      begin
      if opponents[knightorking[i]] = 6 then goto cutoff;
      c := c + 1;
      moves[c] := knightorking[i]
      end
    end of knightorkingmove;

  procedure rookorbishopmove(rookorbishop);
  integer array rookorbishop;
    begin
    for j := 0 step 1 until 3 do
      begin
      k := rookorbishop[j];
      l := rookorbishop[4 × square + j];
      for i := square + k step k until l do
        begin
        if mymanin[i] ≠ 0 then goto newdirection;
        c := c + 1;
        moves[c] := i;
        if opponents[i] ≠ 0 then
          begin
          if opponents[i] = 6 then goto cutoff else goto
            newdirection
          end
        end;
newdirection: end
    end of rookorbishopmove;

  procedure whiteorblackpawnmove(pawn);
  integer array pawn;
```

```
        begin
        for i := 4 × square, i + 1 do
          begin
          j := pawn[i];
          if mymanin[j] ≠ 0 ∨ opponents[j] ≠ 0 then goto
            pawncapture;
          c := c + 1;
          moves[c] := j
          end;
pawncapture: for i := 4 × square + 2, i + 1 do
ep2:    if opponents[pawn[i]] ≠ 0 then
          begin
          if opponents[pawn[i]] = 6 then goto cutoff;
          c := c + 1;
          moves[c] := pawn[i]
          end
        end of whiteorblackpawnmove;
      c := startat;
      for pointer := piece[0] step − 1 until 1 do
        begin
        square := piece[pointer];
        if mymanin[square] = 0 then goto continue;
        moves[c] := pointer;
        moves[c + 1] := mymanin[square];
        moves[c + 2] := square;
        c := c + 2;
        goto moveof[mymanin[square]];
apawn:  if whitemen[square] = 1 then
          whiteorblackpawnmove (wpawn)
        else whiteorblackpawnmove(bpawn); goto nextman;
aknight: knightorkingmove(knight); goto nextman;
aking:  knightorkingmove(king);    goto nextman;
arook:  rookorbishopmove(rook);    goto nextman;
aqueen: rookorbishopmove(rook);
abishop: rookorbishopmove(bishop);
nextman: if moves[c] ≠ square then
          begin
          moves[c + 1] := − square;
          c := c + 2
          end else c := c − 2;
continue: end;
CASTLING: See Appendix 2:
      startat := c
      end of listmoves;

    procedure makemove(pointer, mymen, hismen, mypiece,
      hispiece, continue);
    integer pointer;
    integer array mymen, hismen, mypiece, hispiece;
    label continue;
      begin
      integer from, to;
      goto start;
illegal move: depth := depth − 1; pointer := pointer + 1;
start:
      from := moves[pointer];
      to := moves[pointer + 1];
      if to > 0 then
        begin
ep1:
A1:     if q[depth] = 1 ∧ (to > 56 ∨ to < 9) then
          begin
          pointer := pointer − 1;
          mymen[to] := if mymen[to] = 0 then 5 else
            mymen[to] − 1;
          if mymen[to] = 2 then pointer := pointer + 1;
          end else mymen[to] := q[depth];
A2:     mymen[from] := 0;
A3:     hismen[from] := c[depth];
A4:     c[depth] := hismen[to];
A5:     hismen[to] := 0;
A6:     mypiece[locate[depth]] := to;
ep3:    depth := depth + 1;
        to := numberofmoves[depth] :=
          numberofmoves [depth − 1];
        listmoves(hispiece, hismen, mymen,
          numberofmoves [depth], illegal move);
        locate[depth] := moves[to];
        q[depth] := moves[to + 1];
        backto[depth] := moves[to + 2]
        end else
        begin
B1:     mymen[− to] := q[depth];
B2:     mymen[from] := 0;
ep4:
B3:     hismen[from] := c[depth];
B4:     c[depth] := 0;
B5:     mypiece[locate[depth]] := − to;
        locate[depth] := moves[pointer + 2];
        q[depth] := moves[pointer + 3];
        backto[depth] := moves[pointer + 4];
        pointer := pointer + 3;
        goto continue
        end
      end of makemove;

    procedure reversemove(from, mymen, hismen, mypiece);
    integer from;
    integer array mymen, hismen, mypiece;
      begin
      mymen[backto[depth]] := q[depth];
      mymen[from] := 0;
ep4:  hismen[from] := c[depth];
      c[depth] := 0;
      mypiece[locate[depth]] := backto[depth]
      end of reversemove;

ENTRY: TO READ IN THE TABLES AND PROBLEM
      INSERT APPENDIX 3 HERE WITH APPROPRIATE
      INPUT STATEMENTS:
    c[1] := 0;
    depth := 1;
    numberofmoves[1] := 1;
    listmoves(whitepiece, whitemen, blackmen, numberofmoves[1],
      theend);
    comment When debugging program print the list of moves just
    produced, for example;
    for i := 1 step 1 until numberofmoves[1] do print(moves[i]);
    locate[depth] := moves[1];
    q[depth] := moves[2];
    for w1 := 3 step 1 until numberofmoves[1] do
      begin
      makemove(w1, whitemen, blackmen, whitepiece, black-
        piece, w1continue);
See Appendix 4 in order to: printposition;
      for b1 := numberofmoves[1] + 2 step 1 until numberof-
        moves[2] do
        begin
        makemove(b1, blackmen, whitemen, blackpiece, white-
          piece, b1continue);
        for w2 := numberofmoves[2] + 2 step 1 until number-
          ofmoves[3] do
          begin
          makemove(w2, whitemen, blackmen, whitepiece,
            blackpiece, w2continue);
          n := numberofmoves[4];
          listmoves(whitepiece, whitemen, blackmen, n, not-
            stalemate);
          goto w2continue;
notstalemate: for b2 := numberofmoves[3] + 2 step 1 until
            numberofmoves[4] do
            begin
            makemove(b2, blackmen, whitemen, blackpiece,
              whitepiece, b2continue);
```

```
        depth := 4;
        reversemove(moves[b2 + 1], blackmen, whitemen,
          blackpiece); goto w2continue;
b2continue: depth := 4
        end;
      depth := 3;
      reversemove(moves[w2 + 1], whitemen, blackmen,
        whitepiece); goto b1continue;
w2continue: depth := 3
        end;
    depth := 2;
    reversemove(moves[b1 + 1], blackmen, whitemen,
      blackpiece); goto w1continue;
b1continue: depth := 2
    end;
  printposition; THE ANSWER:
  goto theend;
w1continue: depth := 1
    end;
theend:
    end
```

Appendix 1.1

KNIGHT TABLE

SQUARE									NUMBER OF MOVES
1.	0	0	0	0	0	0	18	11	2
2.	0	0	0	0	0	17	19	12	3
3.	0	0	0	0	9	18	20	13	4
4.	0	0	0	0	10	19	21	14	4
5.	0	0	0	0	11	20	22	15	4
6.	0	0	0	0	12	21	23	16	4
7.	0	0	0	0	0	13	22	24	3
8.	0	0	0	0	0	0	14	23	2
9.	0	0	0	0	0	26	19	3	3
10.	0	0	0	0	25	27	20	4	4
11.	0	0	17	26	28	21	5	1	6
12.	0	0	18	27	29	22	6	2	6
13.	0	0	19	28	30	23	7	3	6
14.	0	0	20	29	31	24	8	4	6
15.	0	0	0	0	21	30	32	5	4
16.	0	0	0	0	0	22	31	6	3
17.	0	0	0	0	34	27	11	2	4
18.	0	0	33	35	28	12	3	1	6
19.	25	34	36	29	13	4	2	9	8
20.	26	35	37	30	14	5	3	10	8
21.	27	36	38	31	15	6	4	11	8
22.	28	37	39	32	16	7	5	12	8
23.	0	0	29	38	40	8	6	13	6
24.	0	0	0	0	30	39	7	14	4
25.	0	0	0	0	42	35	19	10	4
26.	0	0	41	43	36	20	11	9	6
27.	33	42	44	37	21	12	10	17	8
28.	34	43	45	38	22	13	11	18	8
29.	35	44	46	39	23	14	12	19	8
30.	36	45	47	40	24	15	13	20	8
31.	0	0	37	46	48	16	14	21	6
32.	0	0	0	0	38	47	15	22	4
33.	0	0	0	0	50	43	27	18	4
34.	0	0	49	51	44	28	19	17	6
35.	41	50	52	45	29	20	18	25	8
36.	42	51	53	46	30	21	19	26	8
37.	43	52	54	47	31	22	20	27	8
38.	44	53	55	48	32	23	21	28	8
39.	0	0	45	54	56	24	22	29	6
40.	0	0	0	0	46	55	23	30	4
41.	0	0	0	0	58	51	35	26	4
42.	0	0	57	59	52	36	27	25	6
43.	49	58	60	53	37	28	26	33	8
44.	50	59	61	54	38	29	27	34	8
45.	51	60	62	55	39	30	28	35	8
46.	52	61	63	56	40	31	29	36	8
47.	0	0	53	62	64	32	30	37	6
48.	0	0	0	0	54	63	31	38	4
49.	0	0	0	0	0	59	43	34	3
50.	0	0	0	0	60	44	35	33	4
51.	0	0	57	61	45	36	34	41	6
52.	0	0	58	62	46	37	35	42	6
53.	0	0	59	63	47	38	36	43	6
54.	0	0	60	64	48	39	37	44	6
55.	0	0	0	0	61	40	38	45	4
56.	0	0	0	0	0	62	39	46	3
57.	0	0	0	0	0	0	51	42	2
58.	0	0	0	0	0	52	43	41	3
59.	0	0	0	0	53	44	42	49	4
60.	0	0	0	0	54	45	43	50	4
61.	0	0	0	0	55	46	44	51	4
62.	0	0	0	0	56	47	45	52	4
63.	0	0	0	0	0	48	46	53	3
64.	0	0	0	0	0	0	47	54	2

Appendix 1.1*b*

KING TABLE

SQUARE									NUMBER OF MOVES
1.	0	0	0	0	0	2	9	10	3
2.	0	0	0	1	3	9	10	11	5
3.	0	0	0	2	4	10	11	12	5
4.	0	0	0	3	5	11	12	13	5
5.	0	0	0	4	6	12	13	14	5
6.	0	0	0	5	7	13	14	15	5
7.	0	0	0	6	8	14	15	16	5
8.	0	0	0	0	0	7	15	16	3
9.	0	0	0	1	2	10	17	18	5
10.	1	2	3	9	11	17	18	19	8
11.	2	3	4	10	12	18	19	20	8
12.	3	4	5	11	13	19	20	21	8
13.	4	5	6	12	14	20	21	22	8
14.	5	6	7	13	15	21	22	23	8
15.	6	7	8	14	16	22	23	24	8
16.	0	0	0	7	8	15	23	24	5
17.	0	0	0	9	10	18	25	26	5
18.	9	10	11	17	19	25	26	27	8
19.	10	11	12	18	20	26	27	28	8
20.	11	12	13	19	21	27	28	29	8
21.	12	13	14	20	22	28	29	30	8
22.	13	14	15	21	23	29	30	31	8
23.	14	15	16	22	24	30	31	32	8
24.	0	0	0	15	16	23	31	32	5
25.	0	0	0	17	18	26	33	34	5
26.	17	18	19	25	27	33	34	35	8
27.	18	19	20	26	28	34	35	36	8
28.	19	20	21	27	29	35	36	37	8
29.	20	21	22	28	30	36	37	38	8
30.	21	22	23	29	31	37	38	39	8
31.	22	23	24	30	32	38	39	40	8
32.	0	0	0	23	24	31	39	40	5

33.	0	0	0	25	26	34	41	42	5
34.	25	26	27	33	35	41	42	43	8
35.	26	27	28	34	36	42	43	44	8
36.	27	28	29	35	37	43	44	45	8
37.	28	29	30	36	38	44	45	46	8
38.	29	30	31	37	39	45	46	47	8
39.	30	31	32	38	40	46	47	48	8
40.	0	0	0	31	32	39	47	48	5
41.	0	0	0	33	34	42	49	50	5
42.	33	34	35	41	43	49	50	51	8
43.	34	35	36	42	44	50	51	52	8
44.	35	36	37	43	45	51	52	53	8
45.	36	37	38	44	46	52	53	54	8
46.	37	38	39	45	47	53	54	55	8
47.	38	39	40	46	48	54	55	56	8
48.	0	0	0	39	40	47	55	56	5
49.	0	0	0	41	42	50	57	58	5
50.	41	42	43	49	51	57	58	59	8
51.	42	43	44	50	52	58	59	60	8
52.	43	44	45	51	53	59	60	61	8
53.	44	45	46	52	54	60	61	62	8
54.	45	46	47	53	55	61	62	63	8
55.	46	47	48	54	56	62	63	64	8
56.	0	0	0	47	48	55	63	64	5
57.	0	0	0	0	0	49	50	58	3
58.	0	0	0	49	50	51	57	59	5
59.	0	0	0	50	51	52	58	60	5
60.	0	0	0	51	52	53	59	61	5
61.	0	0	0	52	53	54	60	62	5
62.	0	0	0	53	54	55	61	63	5
63.	0	0	0	54	55	56	62	64	5
64.	0	0	0	0	0	55	56	63	3

Appendix 1.2*a*

ROOK TABLE

	E	W	N	S
	1	−1	8	−8
1.	8	1	57	1
2.	8	1	58	2
3.	8	1	59	3
4.	8	1	60	4
5.	8	1	61	5
6.	8	1	62	6
7.	8	1	63	7
8.	8	1	64	8
9.	16	9	57	1
10.	16	9	58	2
11.	16	9	59	3
12.	16	9	60	4
13.	16	9	61	5
14.	16	9	62	6
15.	16	9	63	7
16.	16	9	64	8
17.	24	17	57	1
18.	24	17	58	2
19.	24	17	59	3
20.	24	17	60	4
21.	24	17	61	5
22.	24	17	62	6
23.	24	17	63	7
24.	24	17	64	8

Appendix 1.2*b*

BISHOP TABLE

	NE	SW	NW	SE
	9	−9	7	−7
1.	64	1	1	1
2.	56	2	9	2
3.	48	3	17	3
4.	40	4	25	4
5.	32	5	33	5
6.	24	6	41	6
7.	16	7	49	7
8.	8	8	57	8
9.	63	9	9	2
10.	64	1	17	3
11.	56	2	25	4
12.	48	3	33	5
13.	40	4	41	6
14.	32	5	49	7
15.	24	6	57	8
16.	16	7	58	16
17.	62	17	17	3
18.	63	9	25	4
19.	64	1	33	5
20.	56	2	41	6
21.	48	3	49	7
22.	40	4	57	8
23.	32	5	58	16
24.	24	6	59	24
25.	32	25	57	1
26.	32	25	58	2
27.	32	25	59	3
28.	32	25	60	4
29.	32	25	61	5
30.	32	25	62	6
31.	32	25	63	7
32.	32	25	64	8
33.	40	33	57	1
34.	40	33	58	2
35.	40	33	59	3
36.	40	33	60	4
37.	40	33	61	5
38.	40	33	62	6
39.	40	33	63	7
40.	40	33	64	8
41.	48	41	57	1
42.	48	41	58	2
43.	48	41	59	3
44.	48	41	60	4
45.	48	41	61	5
46.	48	41	62	6
47.	48	41	63	7
48.	48	41	64	8
49.	56	49	57	1
50.	56	49	58	2
51.	56	49	59	3
52.	56	49	60	4
53.	56	49	61	5
54.	56	49	62	6
55.	56	49	63	7
56.	56	49	64	8
57.	64	57	57	1
58.	64	57	58	2
59.	64	57	59	3
60.	64	57	60	4
61.	64	57	61	5
62.	64	57	62	6
63.	64	57	63	7
64.	64	57	64	8

Appendix 1.3*a*

WHITE PAWN TABLE

	+8	+16	+7	+9
9.	17	25	65	18
10.	18	26	17	19
11.	19	27	18	20
12.	20	28	19	21
13.	21	29	20	22
14.	22	30	21	23
15.	23	31	22	24
16.	24	32	23	65
17.	25	17	65	26
18.	26	18	25	27
19.	27	19	26	28
20.	28	20	27	29
21.	29	21	28	30
22.	30	22	29	31
23.	31	23	30	32
24.	32	24	31	65

Appendix 1.3*b*

BLACK PAWN TABLE

	−8	−16	−7	−9
9.	1	9	2	65
10.	2	10	3	1
11.	3	11	4	2
12.	4	12	5	3
13.	5	13	6	4
14.	6	14	7	5
15.	7	15	8	6
16.	8	16	65	7
17.	9	17	10	65
18.	10	18	11	9
19.	11	19	12	10
20.	12	20	13	11
21.	13	21	14	12
22.	14	22	15	13
23.	15	23	16	14
24.	16	24	65	15

25.	33	25	65	34	17	25	18	65
26.	34	26	33	35	18	26	19	17
27.	35	27	34	36	19	27	20	18
28.	36	28	35	37	20	28	21	19
29.	37	29	36	38	21	29	22	20
30.	38	30	37	39	22	30	23	21
31.	39	31	38	40	23	31	24	22
32.	40	32	39	65	24	32	65	23
33.	41	33	65	42	25	33	26	65
34.	42	34	41	43	26	34	27	25
35.	43	35	42	44	27	35	28	26
36.	44	36	43	45	28	36	29	27
37.	45	37	44	46	29	37	30	28
38.	46	38	45	47	30	38	31	29
39.	47	39	46	48	31	39	32	30
40.	48	40	47	65	32	40	65	31
41.	49	41	65	50	33	41	34	65
42.	50	42	49	51	34	42	35	33
43.	51	43	50	52	35	43	36	34
44.	52	44	51	53	36	44	37	35
45.	53	45	52	54	37	45	38	36
46.	54	46	53	55	38	46	39	37
47.	55	47	54	56	39	47	40	38
48.	56	48	55	65	40	48	65	39
49.	57	49	65	58	41	33	42	65
50.	58	50	57	59	42	34	43	41
51.	59	51	58	60	43	35	44	42
52.	60	52	59	61	44	36	45	43
53.	61	53	60	62	45	37	46	44
54.	62	54	61	63	46	38	47	45
55.	63	55	62	64	47	39	48	46
56.	64	56	63	65	48	40	65	47

Appendix 2

The following is inserted at label 'Castling' in the program

```
CASTLING:
  if piece[1] = 5 then goto white king side else
  if piece[1] = 61 then goto black king side else
  goto terminate;
white king side: if white king side castle ∧ whitemen[6] = 0 ∧
    whitemen[7] = 0 then
    begin
    for n := depth − 2 step − 2 until 1 do
    if locate[n] = 1 ∨ locate[n] = 2 then goto white queen
      side;
    end else goto white queen side;
  whitemen[6] := whitemen[7] : = 6;
  n := c;
  listmoves(blackpiece, blackmen, whitemen, n, cant castle wk
    side);
can castle wk side: castle(5, 7, 2, 8, 6);
cant castle wk side: whitemen[6] := whitemen[7] := 0;
white queen side: if white queen side castle ∧ whitemen[2] = 0
  ∧ whitemen[3] = 0 ∧ whitemen[4] = 0 then
    begin
    for n := depth − 2 step − 2 until 1 do
    if locate[n] = 1 ∨ locate[n] = 3 then goto terminate;
    end else goto terminate;
  whitemen[3] := whitemen[4] := 6;
  n := c;
  listmoves (blackpiece, blackmen, whitemen, n, cant castle
  wq side);
can castle wq side: castle(5, 3, 3, 1, 4);
cant castle wq side: whitemen[3] := whitemen[4] := 0;
  goto terminate;
black king side: if black king side castle ∧ blackmen[62] = 0
    ∧ blackmen[63] = 0 then
    begin
    for n := depth − 2 step − 2 until 2 do
    if locate[n] = 1 ∨ locate[n] = 2 then goto black queen
      side;
    end else goto black queen side;
  n := c;
  listmoves(whitepiece, whitemen, blackmen, n, cant castle bk
    side);
can castle bk side: castle(61, 63, 2, 64, 62);
cant castle bk side: blackmen[62] := blackmen[63] := 0;
black queen side: if black queen side castle ∧ blackmen[58] = 0
  ∧ blackmen[59] = 0 ∧ blackmen[60] = 0 then
    begin
    for n := depth − 2 step − 2 until 2 do
    if locate[n] = 1 ∨ locate[n] = 3 then goto terminate;
    end else goto terminate;
  blackmen[59] := blackmen[60] := 6;
  n := c;
  listmoves(whitepiece, whitemen, blackmen, n, cant castle bq
    side);
can castle bq side: castle(61, 59, 3, 57, 60);
cant castle bq side: blackmen[59] := blackmen[60] := 0;
terminate:
```

The following replaces the dummy procedure 'castle' in the program

```
procedure castle(kfrom, kto, rook, rfrom, rto);
integer kfrom, kto, rook, rfrom, rto;
  begin
  moves[c] := 1;
  moves[c + 1] := 6;
  moves[c + 2] := kfrom;
  moves[c + 3] := − kto;
  moves[c + 4] := rook;
  moves[c + 5] := 4;
  moves[c + 6] := rfrom;
  moves[c + 7] := rto;
  moves[c + 8] := − rfrom;
  moves[c + 9] := 1;
  moves[c + 10] := 6;
  moves[c + 11] := kto;
  moves[c + 12] := − kfrom;
  c := c + 13
  end;
```

Finally, the Boolean variables must be declared in the outer block of the program thus

```
Boolean white king side castle, white queen side castle,
        black king side castle, black queen side castle;
```

and initialised suitably on entry depending on which castlings are still valid.

Appendix 3

Algol text to input the relevant contents of Appendix 1 and a two move mate position

```
  for i := 1 step 1 until 9 × 64 do knight[i] := read;
  for i := 1 step 1 until 9 × 64 do king[i] := read;
  for i := 0 step 1 until 4 × 64 + 3 do rook[i] := read;
  for i := 0 step 1 until 4 × 64 + 3 do bishop[i] := read;
  for i := 4 × 9 step 1 until 4 × 56 + 3 do wpawn[i] := read;
  for i := 4 × 9 step 1 until 4 × 56 + 3 do bpawn[i] := read;
CLEAR THE WHITE AND BLACK BOARDS:
  for i := 1 step 1 until 65 do whitemen[i] := blackmen[i] := 0;
NUMBER OF WHITEMEN IS:
  whitepiece[0] := read;
NOW READ IN THEIR VALUE AND POSITION ON
THE BOARD:
  for i := 1 step 1 until whitepiece[0] do
```

```
    begin
    j := read;  k := read;
    whitemen[k] := j; whitepiece[i] := k
    end;
NUMBER OF BLACKMEN IS:
  blackpiece[0] := read;
AND THEIR VALUE AND POSITION ON THE BOARD IS:
  for i := 1 step 1 until blackpiece[0] do
    begin
    j := read;  k := read;
    blackmen[k] := j; blackpiece[i] := k
    end;
  comment The example in Fig. 1 is described by the following data
11
6 10 1 12 5 14 2 18 1 22 1 32 4 33 1 39 3 51 3 57 2 62
7
1  20  1  28  1  30  2  36  5  37  6  38  1  46;
```

Appendix 4

Prints the position, the layout is similar to Fig. 1(*b*)

```
  procedure print position;
    begin
    switch list := p, n, b, r, q, k;
    newline(2);
    for i := 56 step - 8 until 0 do
      begin
      for j := 1 step 1 until 8 do
        begin
        if whitemen[i + j] = 0 then goto printblack else
          writetext('w');
        goto list[whitemen[i + j]];
printblack: if blackmen[i + j] = 0 then goto empty else write-
          text('b');
        goto list[blackmen[i + j]];

p:      writetext('p'); goto sp;
n:      writetext('n'); goto sp;
b:      writetext('b'); goto sp;
r:      writetext('r'); goto sp;
q:      writetext('q'); goto sp;
k:      writetext('k'); goto sp;

empty:  writetext('␣0');
sp:     writetext('␣␣')
        end;
      newline(1)
      end
    end;
```

Note on Algorithm 44

SOLUTION OF NONLINEAR SIMULTANEOUS EQUATIONS

In procedure *nonlinb* the real variable *theta*1 and the arrays *x*1, *x*2 are used in an arithmetic expression before they have had anything assigned to them. This caused an execution error when these procedures were tested using the ICL 1900 Algol compiler.

This difficulty can be avoided by replacing the line before the first call of *inival* by

```
s1 := s2 := theta1 := 0;
for i := 1 step 1 until order do
  x1[i] := x2[i] := 0;
```

Also in procedures *nonlina* and *nonlinb* a semicolon is needed just before the label EXIT. It should be, e.g.

```
F1: type := 1; go to FAIL;
EXIT: end of procedure nonlina;
```

K. Fielding
Computing Centre
University of Essex

Mr. Broyden replies:

The major 'errors' in the algorithm do not, in fact, affect its progress since although *theta*1 and the arrays *x*1 and *x*2 are used before they have anything assigned to them, the initial values they contain are, in fact, just shunted down the stack and eventually disappear off the end; they are not actually used in real calculation. The procedure that I gave you did not fail when I tested it because in those days the ICL Algol translator set all unassigned stores to zero. The policy has now been changed to set all unassigned stores to something which causes overflow. Hence it is necessary to initialise these stores to some arbitrary but finite number and Mr. Fielding chose the logical and obvious value of zero. I hope this will clarify the situation.

Perhaps I should add that the above letter by Mr Fielding was written at my suggestion, was sent to you with my blessing, and received my full approval.

Computing Centre
University of Essex

Note on Algorithm 47

A CLUSTERING ALGORITHM

There is a minor error in the Author's note of Algorithm 47, which does not affect the working of the algorithm. The L_1-condition referred to is not the same as the L_1-condition in Jardine (1969) cited in the paper. The clusters generated by Algorithm 47 satisfy Jardine's L_1-condition but are more homogeneous.

C. J. van Rijsbergen
King's College Research Centre
Cambridge

Note on Algorithm 47

A. H. J. Sale's Note on Algorithm 42 applies equally to Algorithm 47 printed alongside. Algorithm 47 also contravenes the USASI Fortran standard in subroutine CLUST where array Y, having been equivalenced with array Z in the calling program, should not be redefined (see section 8.4.2 in the standard) and again in subroutine PREV where the terminal parameter in a DO statement should not be an expression (section 7.1.2.8). The algorithm also assumes logical unit 0 to be a printing device: it would seem to be better practice in a general routine to pass the output unit number as a parameter.

D. T. Muxworthy
Edinburgh Regional Computing Centre.

[*Editor's apology*. The editor apologises for these errors which were not spotted when the paper was refereed.]

Contributions for the Algorithms Supplement should be sent to
Mrs. M. O. Mutch
University Engineering Department
Control Engineering Group
Mill Lane, Cambridge