Writing Strategy Games on Your Atari®

Techniques for Intelligent Games

FOR ATARI 400, 800, AND XL COMPUTERS

John White

HAYDEN

Writing Strategy Games on Your Atari®

Techniques for Intelligent Games

John White

HAYDEN BOOK COMPANY
a division of Hayden Publishing Company, Inc.
Hasbrouck Heights, New Jersey

First published 1983 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12/13 Little Newport Street,
London WC2R 3LD

Cover illustration by Stuart Hughes

Printed in the United States of America

1	2	3	4	5	6	7	8	9	PRINTING
84	85	86	87	88	89	90	91	92	YEAR

CONTENTS

For Janet

Preface

There are very few books on writing computerised games of strategy and fewer still giving actual worked examples. This book aims to fill that gap for owners of Atari home computers, although I hope that the owners of other machines will also find it useful.

This is a book pitched at intermediate level. It assumes that the reader is already acquainted with Atari BASIC. Machine code programmers have also been catered for, with many hints on how to modify routines for their programs. These sections are interwoven with the rest of the text and BASIC programmers can lightly skip such paragraphs, knowing that they are not missing any vital information.

The programs that are presented here generally require less than 16K of free RAM for their operation; the exception is Warp Trog which requires 32K. The Atari 400 and 800 models will also require the Atari BASIC cartridge; the more recent models 600XL and 800XL have Atari BASIC fitted as standard.

The programs are written to illustrate principles rather than to be 'clever'. The knowledgeable reader is encouraged to enhance them.

I make no apology for numerous references to commercial strategy programs, many unavailable for Atari machines, since much can be learned by examination of the state of the art.

For the same reason, many of the underlying principles in each chapter are illustrated by reference to chess. Although BASIC is not suitable for programming chess, the game provides good illustrations of all the points made in the book, while at the same time the rules are well known.

Writing strategy games requires the blending of many sciences; not only a knowledge of programming but also of certain mathematical and coding techniques. In order to keep the book to a manageable length, I have given a brief outline of such methods where appropriate, then referred readers to other sources for more detail.

Acknowledgements

I am grateful to Brendon Gore of *Popular Computing Weekly* who suggested that I should write this book, and to my wife who spent many long hours typing it into a word processor. I also thank Norman Lattimer,

who took great trouble to read through the text.

Any book of this sort must acknowledge its debt to David Levy, former chess International Master and now a programmer of strategy games, whose books and other writings have influenced so many aspiring program writers.

KEY

The following standard variables are used in programs in this book:

Q : Evaluation score
QQ : Best score yet found at 1-ply
X,Y : Present location of piece on board at square A(X,Y)
U,V : Temporary new location of piece moving to square A(U,V)
AO : +1 when computer is considering program's best move
− when computer is considering opponent's reply

The mathematical expression 10 exp N in the text is equivalent to the number 10^N. Thus 10 exp 33 is equivalent to 10^{33}, i.e., 1 followed by 33 zeros: 1,000,000,000,000,000,000,000,000,000,000,000.

CHAPTER 1
Introduction

There comes a time in any computer owner's life when he becomes tired of zapping aliens. As the 1,079,864th flying saucer crashes spectacularly to the ground (or, more often, the player crashes instead), he thinks about signing a peace treaty with the rest of the galaxy and turning to something else.

The next option that most computer owners try is one of the many adventures that are available in machine code or BASIC. Sooner or later, even this palls (or the owner doesn't like solving puzzles), leaving the last, and most difficult option, the field of intelligent games.

There are a huge number of games for computers where 'the other side' is trying to overcome you, just as you are trying to overcome it. The enemy moves may be divided arbitrarily into *random* moves, *directed* moves and *intelligent* moves.

An example of a random enemy move occurs when you are flying a spaceship, or steering a submarine, when an enemy spaceship/destroyer suddenly pops up and starts firing. What has happened is that the program has encountered a line such as:

```
1000 Q = INT(RND(1)*6+1): IF Q > 5 THEN GOSUB ATTACK
```

where Q is a variable set to a figure between 1 and 6. If it exceeds 5, then the enemy will start its attack.

Directed moves give the appearance of being intelligent, since they are aimed directly at you, the player. As an example, a spaceship will always fire exactly at you, instead of firing randomly in all directions. Again, the spaceship may also make its move directly towards you (with the occasional random movement to make life harder). This is achieved by continually reducing the distance between both pieces. Let us suppose that your spaceship is positioned on the screen at location P,Q and the enemy is located at position X,Y. Then the enemy spaceship can steer towards your spaceship with the simple routine:

```
1000 IF X > P THEN X = X - MO
1010 IF X < P THEN X = X + MO
```

1020 IF Y > Q THEN Y = Y – MO
1030 IF Y < Q THEN Y = Y + MO

where MO is the number of screen locations that it can move in any one turn.
Space Invaders (TM) is a classic example of a game combining firing at random times with directed angle of firing, at your cannon. A trivial example of the same thing can be seen in **Program 1-1:**

```
5 REM PROGRAM 1-1
10 GRAPHICS 17:POKE 752,1
20 X=10:FIRE=100
30 COLOR 3:PLOT 10,1
40 COLOR 0:PLOT X,18
50 IF STICK(0)=11 THEN X=X-1:IF X<2
   THEN X=2
60 IF STICK(0)=7 THEN X=X+1:IF X>18
   THEN X=18
70 COLOR 43:PLOT X,18
80 IF RND(1)>0.96 THEN GOSUB 100
85 IF STICK(0)=15 AND RND(1)>0.7 THEN
   GOSUB 100:REM FIRE MORE OFTEN IF
   STATIONARY
90 GOTO 40
100 COLOR 42:PLOT 10,2:DRAWTO X,17
110 COLOR 0:PLOT 10,2:DRAWTO X,17
120 POSITION X,18:PRINT #6;"BANG"
130 SOUND 0,40,8,8
140 FOR K=1 TO 100:NEXT K
150 SOUND 0,0,0,0
160 POSITION X,18:PRINT #6;"     "
170 RETURN
```

Plug a joystick into port 1, type in **Program 1-1** and RUN it, moving your 'ship' – the character '+' – from left to right with the joystick. The 'enemy' – the character '#' – will randomly fire shots directly at your ship, wherever it moves. If you stand still, you will be hit more often.

Problem: Modify **Program 1-1** so that the enemy moves towards you, instead of firing. (Hint: use MO = 1).

An interesting example of directed play was the mechanical device made by a Spaniard, Quevedo, in 1890 for the express purpose of playing

the king-rook versus king ending at chess. The result should always be a forced win for the side with the rook, and it should take no more than 16 moves at most, with best play.

According to a reconstruction of the algorithm by Michie in 1975, Quevedo's machine made no intelligent moves, but directed its pieces towards a win by the following sequential process:

1) If the rook is threatened by EK (Enemy King), move it to the furthest file from the EK

Else 2) If the vertical distance between EK and rook > 1 square, move rook 1 square down

Else 3) If vertical distance between king and EK > 2 squares, move king 1 square down

Else 4) SPACE = horizontal distance between king and EK. If SPACE = 0 then move rook down 1 square (check or checkmate move)

Else 5) If SPACE is odd then move rook 1 square horizontally towards centre of board

Else 6) (SPACE is even). Move king 1 square so as to reduce SPACE.

Note that the program does not at any stage have to pick between several moves. Its move is always forced by the program sequence.

Quevedo's program was a simple one and always assumed that the pieces started on different ranks (try it with king and rook on the same rank) and that the enemy king was always nearest to the bottom of the board, while the rook was the next nearest to the bottom. It sometimes took 62 moves to force checkmate, but it was the first program to be able to play any part of chess at all.

Intelligent moves are harder to program, and this book is devoted solely to telling you how. We define an 'intelligent' move as one which the program selects after considering a number of different alternatives. It is as though, in **Program 1-1,** the 'enemy' tested several different shots before firing the one which hit your 'ship'.

An intelligent game is one which uses intelligent moves. Note that even a comparatively trivial game like noughts and crosses (tic-tac-toe) or hexapawn can rate as an intelligent game by this definition.

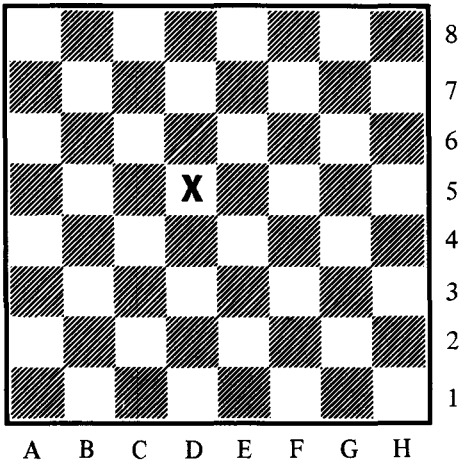
Most intelligent games are strategy games played on two-dimensional boards, such as chess on a chess board.

To understand much of this book it is absolutely essential to know how to use arrays. An account is given in the Atari BASIC book but a brief description will be given here.

An array consists of a large number of locations which are distinguished by a coordinate system. As with reading a map, the board is divided into grids.

A chess board is numbered as a grid, as shown in **Diagram 1-1**, so that the bottom left square is labelled (A,1), the top right is (H,8) and square X is (D,5). The Atari computer will not recognise an array of letters and

Diagram 1-1



numbers, so the letters must first be translated into numbers. This is easily done by turning each letter into its number in the alphabet; thus A = 1, B = 2, C = 3, ..., Z = 26.

It can be represented in BASIC by

```
10 NUMBER = ASC(NAME$) - 64
```

where NAME$ is the letter which you are trying to convert to a number, and NUMBER is the result.

So chess location (A,1) becomes (1,1) in Atari internal code.

The Atari 400 and 800 computers, and their related successors, are fine machines with excellent sound and colour graphics capabilities. Atari BASIC is available as a cartridge for the 400 and 800 machines and is built into the new range of models. They do, however, have a number of idiosyncracies not commonly found in other machines. The most significant example of this is their handling of string arrays. The very common Microsoft BASIC – which is also available as a cartridge for the Atari computers – permits multi-dimensional string arrays such as A$(5,5). Standard Atari BASIC allows only one-dimensional string arrays, such as A$(25). This makes conversion from Atari BASIC to an assembler routine easier than Microsoft's approach, but it is otherwise much less flexible.

Another weakness of Atari BASIC is that all arithmetic is done in slow, memory-expensive six-byte BCD floating-point format, and there is no facility for using fast, two-byte integer arithmetic. By way of compensation, Atari BASIC's method of tokenising BASIC lines as soon as you enter them makes Atari BASIC run faster than many Microsoft

BASICS. Curiously, though, Atari Microsoft BASIC, which does not tokenise lines until the program is RUN, is actually faster than standard Atari BASIC!

Useful features of Atari BASIC include an error trapping mode, called simply the TRAP statement. The hardware provides a real-time clock, useful in many games, which can also be called in BASIC by PEEKing memory locations 18, 19 and 20, the last location being updated every 1/60th of a second. Type in **Program 1-2** and see.

```
5 REM PROGRAM 1-2
10 POKE 18,0:POKE 19,0:POKE 20,0
20 GRAPHICS 0:POKE 752,1
30 TIME=256*256*PEEK(18)+256*PEEK(19)
   +PEEK(20)
40 POSITION 8,10:PRINT "ELAPSED TIME =
   ";INT(TIME/60);" SECONDS    "
50 GOTO 30
```

The highest value which can be accommodated in Program 1-2 is 279620 seconds, when it resets to zero. This is more than three days, long enough for virtually all applications.

Machine code programmers are well catered for. The central processing unit (CPU) is the well tried and tested 6502 chip running at 1.79 MHz (USA) or 2.217 MHz (UK). There is an Atari assembler editor on cartridge – mostly designed for adding machine code routines to BASIC – and a Macro Assembler on disk, as well as several independent offerings.

Other languages offered include the newly trendy Forth, C, Lisp, Pascal and a BASIC Compiler, but not, to my knowledge, the old stalwart Fortran.

It is clear that the Atari computers are extremely flexible, so that there is no difficulty in programming excellent intelligent games once the principles have been mastered.

One feature that all intelligent games have in common is that they need to run quickly. Since many moves may be rejected before the right one is found, every effort should be made to accelerate the essential routines.

The seminal book *De Re Atari* (APX publishers) has a whole section on increasing speed. Top of the list is RECODE the program. Only further down do we find such well-known techniques as stacking the most commonly called routines at the head of the program (BASIC searches through a program, looking for a called line number, from top to bottom) and using variables instead of numbers. For example, instead of saying repeatedly

Q = 1 + 2 * 3

you should write

Q = A + B * C

where A, B and C were previously defined at the beginning of the program as being equal to 1, 2 and 3.

Recording can be tremendously effective in accelerating speed. Some time ago I copied a program (one of my own!) from another computer onto my Atari. The lines were intended to display a board on the screen, and translated as follows, in **Program 1-3:**

```
5 REM PROGRAM 1-3
10 GRAPHICS 0:POKE 752,1
20 FOR I=1 TO 30:FOR J=1 TO 20
30 PLOT I+8,J+2
40 NEXT J:NEXT I
50 POSITION 0,0
```

When I ran this program, a board of hearts appeared. I consulted my new Atari manual, had a brainwave, and entered in the following:

30 COLOR 43: PLOT I+8, J+2

Type in the modified program and RUN it. Success! A board (of '+' symbols) appears on the screen.

Now list the program. Note that the statement COLOR 43 is needlessly evaluated 599 times. No wonder the map takes six seconds to appear in full.

One can recode Program 1-3 entering the lines below:

15 COLOR 43
30 PLOT I+8, J+2

Re-RUN the program. It takes five seconds to display the board, one second less than the earlier version.

We can recode Program 1-3 again, to give **Program 1-4:**

```
5 REM PROGRAM 1-4
10 GRAPHICS 0:POKE 752,1
15 COLOR 43
20 FOR I=3 TO 22
30 PLOT 8,I:DRAWTO 38,I
```

```
40 NEXT I
50 POSITION 0,0
```

Enter **Program 1-4** and RUN it. The board now appears in one second. The last square is always missing, and needs to be filled in with

45 PLOT 38,22

Note that only one FOR-NEXT loop is now used.

Problem: What would happen if you replaced line 20 in Program 1-4 with

20 FOR I = 8 TO 38

and replaced line 30 with

30 PLOT I,3: DRAWTO I,22

Try it and see, with an accurate stop watch.

Recoding has reduced the board display time from six seconds to one second. The board display can also be done in machine code, **Program 1-5.**

```
5 REM PROGRAM 1-5
10 DIM E$(42):CHECKSUM=5790:C=0
20 FOR I=1 TO 41:READ B:POKE ADR(E$)+I,B:
   C=C+B:NEXT I
30 DATA 104,162,0,160,0,104,133,205,104,
   133,204,169,11,145,204,200,192,30,208,24
   9,160,0,232,224
40 DATA 20,240,13,24,169,40,101,204,133,204,
   144,231,230,205,176,227,96
50 IF C<>CHECKSUM THEN PRINT "ERROR IN DATA
   ":STOP
60 SCR=256*PEEK(89)+PEEK(88)+129
70 GRAPHICS 0:POKE 752,1:Q=USR(ADR
   (E$)+1,SCR)
80 POSITION 0,0
```

Ignoring the time spent setting up the machine code program in string E$, the board display appears virtually instantly. Yet, to the human eye, the reduction of time from six seconds to one second in BASIC seems to be much more important than the further reduction from one second to zero seconds.

Coding in assembly language can be very useful, and we shall see in a later chapter that it is essential in many circumstances, but never forget

that BASIC will run sufficiently fast for many intelligent games if it is properly coded.

A fashionable method of programming computers in high-level languages these days is called *structuring*. Essentially this means deploying the whole program as a series of subroutines, all placed in sequence and called by a short main program which contains practically nothing else. Even the initialisation procedures are stored in an initialisation subroutine.

Practitioners of structuring are very fond of their art, which they seem to see as an end in itself, and look down their noses at those who do not structure their programs.

Naturally, I would advise readers to place their program routines in easily distinguished blocks. However, true structuring runs counter to the principle of having the most-called routines at the top of the program and has little else to commend it. I would emphasise that structuring programs is only of benefit to the programmer, and not to the user. Ultimately, one hopes that a user will be the end product of a program, not another programmer.

CHAPTER 2
The Evaluation Function

Underlying all games of strategy, and indeed most of life, is the concept of assigning a score to each of a number of possible moves, then picking the best of the alternatives.

A simple, everyday example of this occurs when we have $300 to spend on a number of glittering gadgets. We could spend all of it on (a) a new Atari 400 (TM) computer, or (b) $200 on a new fridge and $100 on a transistor radio or (c) $300 on a new television, or (d) $300 on 900 Mars Bars.

Which choice we make depends on the value or score we assign to each possibility. Let us say that our test of the various possibilities is to score each one according to how many hours satisfaction each will give us, counted in months we shall spend with them. Then we can assign the following scores:

a) Atari 400	100 (at least!)
b) Fridge and radio	10 (never use them)
c) Television	20 (evenings only)
d) 900 Mars Bars	0.5 (eat them 'til I'm sick)

On this basis, clearly the Atari has the best score, and therefore should be the one that we spend our money on.

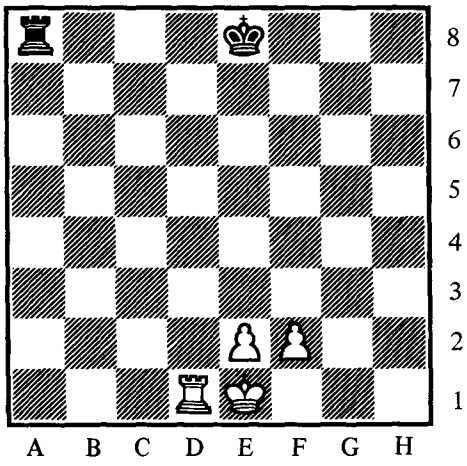
In any game of strategy, it should be possible to assign a score to the position which arises after any move.

The game of chess provides an excellent example. Chess is one of a number of two-player strategy games known as *zero sum* games. In an equal position, the score is evaluated as zero. If the position is not equal, it is as bad for one side (−N) as it is good for the other (+N).

Chess players over the years have evolved a series of scores for each chess piece used, reflecting the relative values of each. A pawn is worth 1 unit, a bishop or knight 3 units, a rook 5 units and a queen 9 units. A king must not be captured, or the game is lost, so we assign it a very high score such that no combination of other pieces will exceed its value. I shall give the king a value of 255 units, for reasons which I shall explain later (Chapter 5).

In **Diagram 2-1,** we can evaluate the position as +255+5+1+1 for white, the computer, and +255+5 for black, the opposition. By

Diagram 2-1



convention, a positive score is always taken as favourable for the program, so it would score

White pieces – Black pieces
= (255+5+2) – (255+5)
= +2

The program evaluates the diagram as indicating that it is winning by 2 units.

How would we evaluate Diagram 2-1 in BASIC? The pieces are all placed on a board which can be represented by an array A(8,8).

The white pieces are located at squares e1 (king), d1 (rook) and e2 and f2 (pawns). We write A(5,1)=255: A(4,1)=5: A(5,2)=1: A(6,2)=1. Remember that we convert letters to board numbers.

Similarly, for the black pieces at e8 and a8, we write A(5,8)=−255: A(1,8)=−5. The rest of the board consists of zeros.

The following subroutine will evaluate the chess board:

Program 2-1

```
1000 Q=0: FOR I=1 TO 8
1010 FOR J=1 TO 8
1020 Q=Q+A(I,J)
1030 NEXT J : NEXT I
1040 RETURN
```

Use this calling program:

```
10 DIM A(8,8,)
20 FOR I=1 TO 8: FOR J=1 TO 8
30 A(I,J)=0
40 NEXT J : NEXT I
50 A(5,1)=255 : A(4,1)=5: A(5,2) =1: A(6,2) = 1
60 A(5,8) = −255: A(1,8) = −5
70 GOSUB 1000
80 PRINT "EVALUATION = "; Q
90 STOP
```

Type in lines 10 to 1040 and RUN. The result should be +2.

Examine the subroutine carefully. The variable Q is used to store the evaluation score which is obtained, and it *must* be cleared (set to zero) before use.

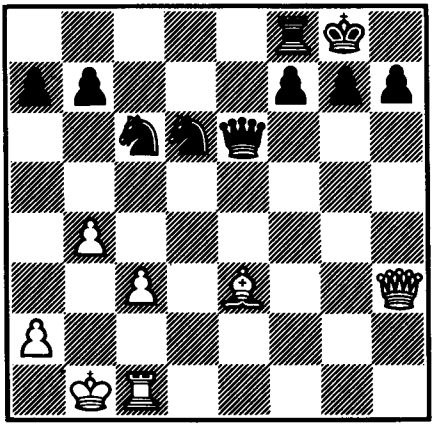
In line 1020, the value of each board square is added, or subtracted if negative, from the total stored in Q.

There is no reason why we should use Q to keep the evaluation, but I have done this for many years now, and it will be used for this purpose in the rest of the book.

The example I have just given for the chess position is known as the *material count* for the chess position. In chess it is necessary to add many other factors (such as mobility and control of the centre) to get a sensible evaluation score, but for many other games the material count will suffice. Try putting in other chess values to Program 2-1 and see the results.

Now try setting up your own chess board, with pieces scattered over it, and evaluate the score. As an example, try the position in **Diagram 2-2.** I make the score −5; this means that the computer is losing by a score equal to a rook (it is actually losing by a knight and 2 pawns).

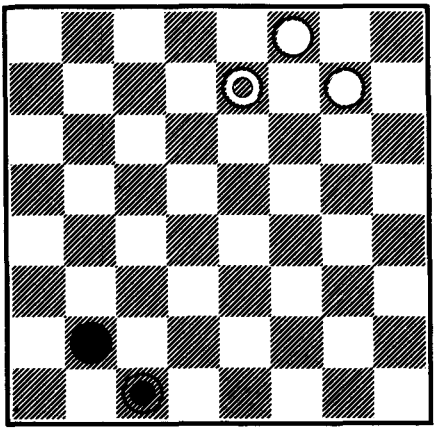
Diagram 2-2



The means of evaluating a position in a program is known as the *evaluation function,* abbreviated to EF.

Problem: Modify **Program 2-1** so that it will evaluate the material count of the draughts (checkers) position shown in **Diagram 2-3.**

Diagram 2-3



○ = white man
◎ = white king
● = black man
◉ = black king

Use +1 or −1 for a white or black man and +3 or −3 for a white or black king. RUN **Program 2-1.** I make the result +1.

Let us modify **Program 2-1** to take account of pawn movement. We know that if a pawn reaches the eighth rank, then it becomes a queen. So, we can assume that advancing a pawn is a good thing. I am adding a score of 0.01 to the evaluation score (Q) for every row that the pawn has advanced. 0.01 is an arbitrary figure which will not upset the material count too much.

If we look at **Diagram 2-2** again, there is one white pawn on row 2, so we add 0.01*2. There is one pawn on row 3, for which we add 0.01*3 and there is one pawn on row 4, so we add 0.01*4 for this. In general, we add 0.01*J for each pawn on the Jth row.

There are five enemy pawns on line 7, so we subtract 0.01*(9–7) for each. (Why 9–7?) In general, we subtract 0.01*(9–J) for each pawn on the Jth row.

Type in the following lines to **Program 2-1:**

```
50 A(1,7) = -1: A(2,7) = -1: A(6,7) = -1: A(7,7) = -1: A(8,7) =
-1
55 A(6,8) = -5: A(7,8) = -255: A(3,6) = -3: A(4,6) = -3: A(5,6) =
-9
60 A(1,2) =1: A(2,4) =1: A(3,3) =1: A(2,1) =255
65 A(5,3) =3: A(8,3) =9: A(3,1) =5
1025 IF A(I,J) =1 THEN Q=Q + J*0.01
1026 IF A(I,J) = -1 THEN Q=Q - (9-J)* 0.01
```

RUN **Program 2-1.** The new evaluation score should be −5.01.

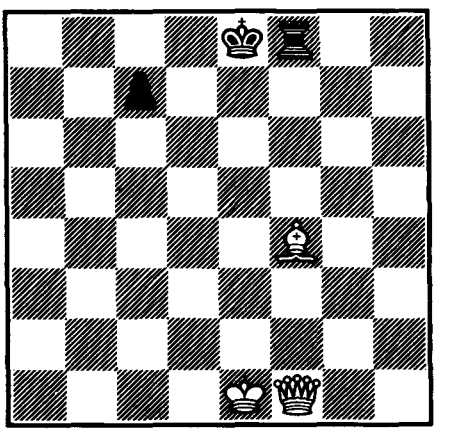
So far, we have considered only evaluating positions which arise after we have made our move. We take each piece which can be moved, temporarily move it to the new square, evaluate the new position (remembering to temporarily remove any captured piece), then replace the piece in its old position (and replace any captured pieces).

Another possibility which is sometimes used is to evaluate the worth of moving a piece, without evaluating the full position that then arises.

This is obviously much faster than evaluating the complete position, but it misses the full value of the position. In the special case of *material count* only, the results are identical, but strategic factors may be overlooked.

For example, if we move a bishop to capture an enemy pawn, we can score the move with the value of the pawn. What the program has missed, though, is that moving the bishop has uncovered an attack on the program's queen. See **Diagram 2-4.**

Diagram 2-4



A full EF would spot the attack on the queen (if pins were evaluated). It is possible to program some protection against this kind of oversight,

but the results never – in my experience – match the full EF. The move-evaluation becomes unwieldy and ultimately, if taken to excess, the original time saved is lost on testing the strategic consequences of the move.

Evaluating *moves* in place of *positions* is generally best suited to tactical games such as draughts, not strategic games like chess.

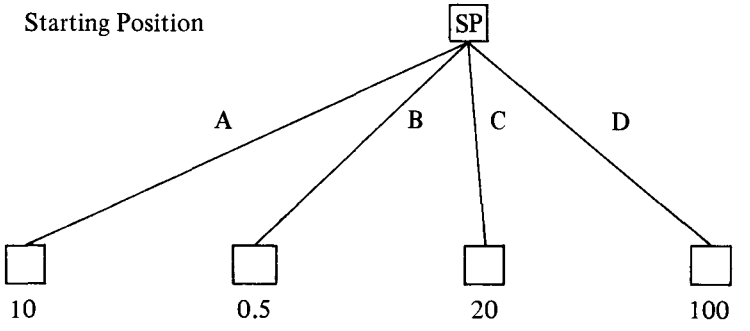
Finding the best move

As the program runs, it will test the move of each piece it has, scoring each move according to the EF. Often, one piece will have several moves, with one score for each.

This can be represented by what is called a *tree,* which, unusually, has *branches* below the surface instead of above (perhaps they should be called roots?).

The starting position (SP) can be scored by the EF before evaluating all the other moves, although this is not necessary in the example following, and all the other moves are scored as shown in **Diagram 2-5.**

Diagram 2-5



Move A results in a score of 10, move B results in a score of 0.5, and so on.

As the program makes each move and obtains the score (Q), it compares this score with another store (QQ) which is set to some suitably low figure so that all possible subsequent scores are higher than its original value.

Theoretically, QQ should be set at –infinity. Atari BASIC does not like –infinity, so we shall assign QQ the value of –1000. This is lower than the material count in chess if the program loses its king (255), nine queens (9*9 = one for each promoted pawn and the original queen), two rooks (2*5), two bishops (2*3) and two knights (2*3).

If the score Q for the move is higher than QQ – which it must be for the first move – then QQ is exchanged for the value of Q, and other stores

are used to hold all the data concerning the move of the piece which led to that score; where it moved from (two locations) and where it moved to (two locations). Store QQ now contains the score of the first move.

Each subsequent move score is compared with the new value in QQ, and, if it is larger, again QQ is exchanged for the new score, and the move data is updated to the new move.

An example will make the method clearer:

Program 2-2

```
10 DIM X (20),Y(20),U(20),V(20),Q(20): QQ=-1000 : EVALUATE
=1000
20 FOR I = 1 TO 20
30 GOSUB EVALUATE : Q(I)=Q : X(I)=I : Y(I) = I : U(I) = I :
V(I) = I
40 IF Q(I)>QQ THEN QQ=Q(I):X=X(I):Y=Y(I):U=U(I):V=V(I)
50 NEXT I
60 PRINT QQ,X;Y,U;V
70 STOP
80 FOR I=1 TO 20
90 PRINT Q(I),X(I);Y(I),U(I);V(I)
100 NEXT I
110 END
1000 REM EVALUATE
1010 Q=0:Q=Q+RND(1)
1020 RETURN
```

The evaluation in **Program 2-2** consists solely of a random number generator, and the 'moves' are the numbers 1 to 20.

RUN Program 2-2. It will print a value for QQ along with values for X,Y,U and V which are the values associated with QQ. Remember that QQ contains the best score of all 20 moves. The program will also display

```
STOPPED AT LINE 70
```

Make a note of the printed variables QQ,X,Y,U and V. Then type in CONT (return).

The program will continue by printing out all the random numbers that were evaluated, together with all the associated 'moves'.

Check carefully that the value you wrote down for QQ is the highest random number present (it is possible, but unlikely, that some other numbers will equal QQ, but none will be higher). Check, too, that the values for X(I),Y(I),U(I) and V(I) corresponding to the highest value of Q(I) match the values of X,Y,U and V which you wrote down.

If they don't match, then either you have made a typing or writing error, or you have not yet noticed that there is another value of Q(I) which matches QQ.

This method is used to pick the best move for the program out of any number of possibilities.

Sometimes we want to know what the worst move is for the program, i.e., the best move for the opposition. In that case, we set the store QQ equal to + infinity, let us say QQ = + 1000, and line 40 in Program 2-2 is replaced by:

```
40 IF Q(I) < QQ THEN QQ = Q(I): X=X(I): Y=Y(I): U=U(I):
V=V(I)
```

The astute reader may have wondered why, if the first move by the program always serves to replace QQ with the score of that move, the program does not start with

QQ = (score for first move)

instead of

QQ = − infinity (−1000)

The reason is simply programming convenience. Compare the following:

```
10 QQ = −1000
20 (MOVE)
30 (EVALUATE)
40 IF Q(I) >QQ THEN QQ = Q(I): etc.
50 GOTO 20
```

and

```
20 (MOVE)
30 (EVALUATE)
35 IF (Move No. = 1) THEN QQ = Q(1)
40 IF Q(I) > QQ THEN QQ = Q(I): etc.
50 GOTO 20
```

The first program considers the redundant line 10 just once. The second considers the equally-redundant line 35 for every move – and that takes time!

Random selection between moves of equal merit

The method of move selection in **Program 2-2** will give the best of several alternative moves. If there are several moves which are equally the strongest, then the first one will be taken.

If line 40 is rewritten as

```
40 IF Q(I) > = QQ THEN QQ = Q(I): etc.
```

then the last of the equal strongest moves will be selected.

In either case, the play will never vary. Often this is of no great significance, especially if there are a large number of alternative moves and the evaluation function is complex.

In other cases, where there is a simple EF, it is useful to enable the program to select randomly between the several strongest moves. This is most easily done by adding a small random number to the EF. For example, insert into Program 2-1

```
1035 Q = Q + INT(RND(1)*3+1)
```

The line adds a random number between 1 and 3 to the evaluation score.

If the random number is sufficiently large, then the program will be able to select between several nearly-best moves as well as the best move; convenient with complicated EFs, which give different scores to several moves all of which are nearly as strong as each other.

Commercial programs for strategy games vary as to whether they add a randomising factor. Sci-Sys chess computers generally do not, for example, whereas Fidelity chess computers usually have a wide variation between moves (large random factor), and computers from Applied Concepts a lesser variation. In any case, the latter machines make it possible to leave out the random factor, if it is unwanted.

Adding random numbers to an EF poses no problem in BASIC, thanks to the RND command. With machine code programs, it is necessary to PEEK into the ROM location of 53770. The latter has access to part of the POKEY chip and takes the most significant byte from the polynomial counter, giving a random eight bit number between 0 and 225 (decimal). This location is updated at machine code speeds, so that successive calls to 53770 from a machine code routine will give continually different random results.

CHAPTER 3
Search in Depth

In the last chapter, we considered how the program finds its best move. However, before making any move, it is always wise to consider the consequences.

If a chess program decides that it should capture a pawn with its queen, it must also decide whether the opponent will in turn capture the queen – or capture another major unit elsewhere on the board. The original score for capturing the pawn (+1) can be turned, after loss of the queen, into a score of (+1−9)=−8, leading to an undesirable position according to the material count EF.

The program tries to make moves which will maximise its evaluation score; it is only natural that the opponent will make counter-moves which will tend to maximise his score; that is, by minimising the program score. In turn the program will try to make yet further moves which will reduce the damage that the opponent can do to the program's score. This process will go on as deep as we look into the projected sequence of moves.

In general, suppose that the program can make three moves – A,B, and C – and that the opponent can make three different responses to each move A,B and C. The positions arising after each opponent counter-move need to be scored.

The program may initially score each move A, B and C as +4, +5, and +2, so that move B looks the best. However, if the positions arising from the three counter moves by the opponent to move A are scored as 6,−1 and 1, then the best that the program can hope for from A is a score of −1, since the opponent will – or should – play the move which is worst (lowest scoring) for the program. The score of −1 is known as the 'backed-up' score for move A.

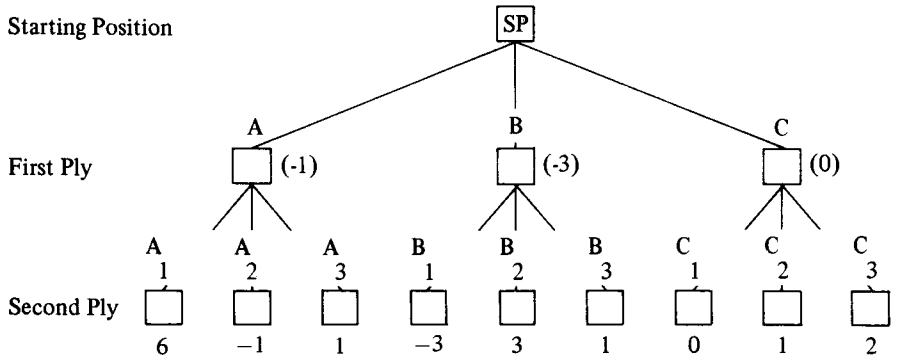
Similarly, if the three counter-moves from move B give positions scored at −3, 3 and 1 then the best score the program can hope for from move B is −3.

Problem: If the counter responses to move C lead to positions scored as 0, 1 and 2, what is the 'backed-up' score for move C? (0)

It is helpful to draw these moves in the form of a tree, as in **Diagram 3-1.**

Each junction between moves is known as a 'node'. The three nodes resulting from moves A, B and C represent the first move looked forward

Diagram 3-1



by the program, and is called the 'first ply' of search.

Each move A, B and C has a further 3 nodes, representing the opponent's three counter-moves. These 9 nodes represent the second move looked forward by the program, (one move for the program, one move for the opponent), and are called the 'second ply' of search.

Note that the 'backed-up' scores for moves A, B and C, respectively −1, −3 and 0, mean that move C is now the best for the program.

The program can look further forward indefinitely. Each second ply node can be sub-divided into several third ply nodes, representing moves by the program to modify the opponent's moves. Each third ply node can be composed of a number of fourth ply nodes.

An example of such a deep tree is given below **Diagram 3-2.**

At the fourth ply, the opponent is trying to minimise the score, so the backed-up score from moves A111 to A113 is +1 at the third ply; from moves A121 to A123 it is −1. At the third ply, the program is trying to maximise the score, so the backed-up second ply score from 1, −1 and 0 is 1.

At the second ply the opponent is trying to minimise the score, so the backed-up first ply score from 1, 2 and 3 is 1.

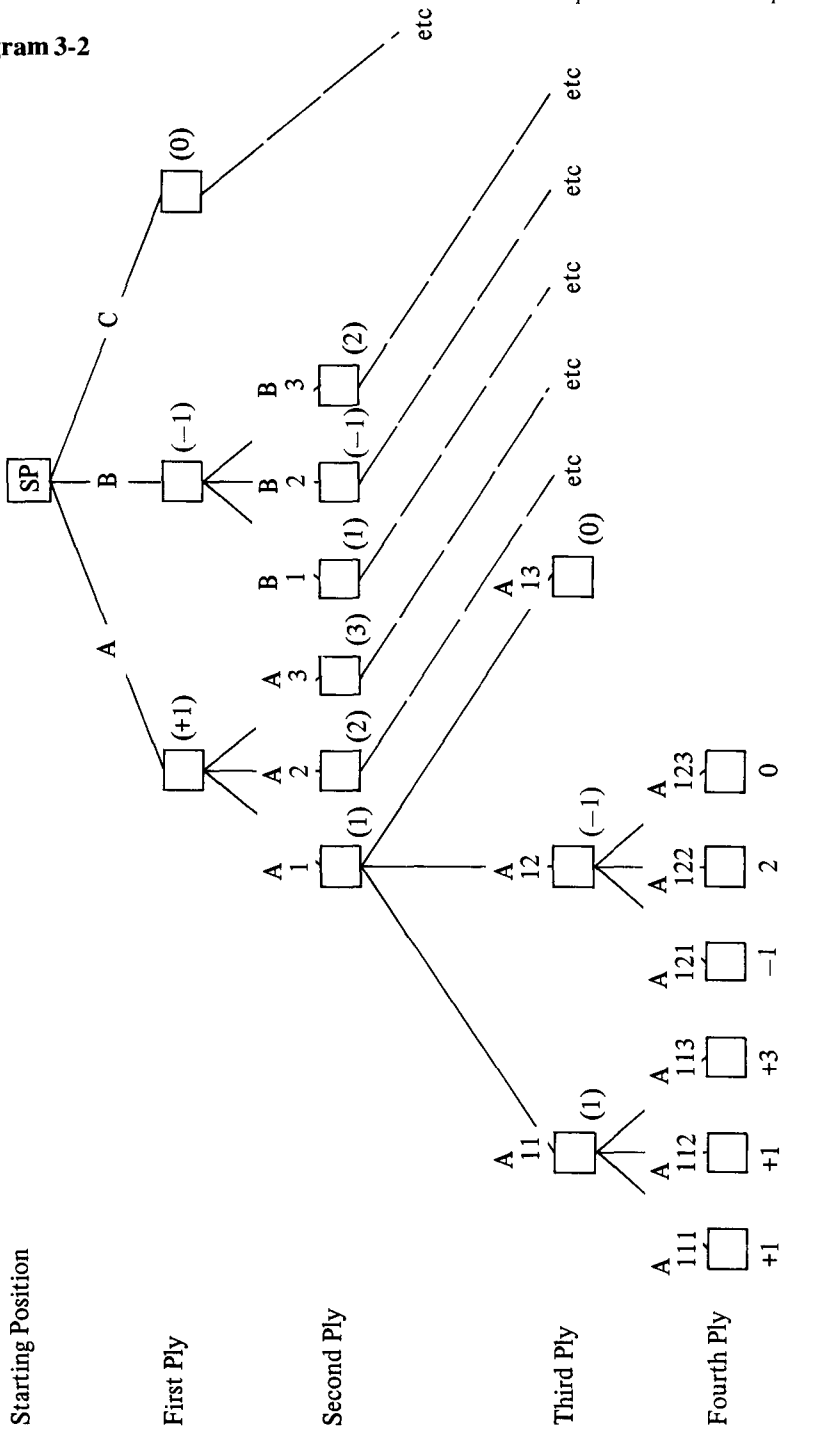
At the first ply, the program tries to maximise the score, so the actual move taken from A (1), B(−1) and C (0) will be move A.

This process for finding moves, by maximising the program score whilst minimising the opponent's response, is known as the *minimax* method.

Minimax searching is used not only in games of strategy, but also in the Pentagon's (and Moscow's?) war games. Both sides try constantly to take actions which benefit their own side at minimum cost. Think about that next time there are missiles in Cuba or battlefleets off Nicaragua.

The method of searching all legal moves and replies to examine which is the best is known as a 'full-width' search. A Selective Search uses a complicated EF to pick out the most plausible moves for further

Diagram 3-2


Starting Position
First Ply
Second Ply
Third Ply
Fourth Ply
SP
A
B
C
(+1)
(−1)
(0)
(3)
(2)
(1)
(−1)
(2)
(1)
(0)
(−1)
(1)
+1
+1
+3
−1
2
0
111
112
113
121
122
123
11
12
13
1
2
3
etc

examination in depth, while rejecting the rest. Selective search is normally carried out at deeper levels after an initial full width search at the first ply.

How far ahead should a program search? If you study a game 'tree', you will see that the number of positions to score increases geometrically the deeper you search. If we assume that both sides always have 10 moves each, then at depth 1 (first-ply) the computer will score 10 positions. At depth 2 (2-ply) it will score 100 positions, at depth 3 (3-ply) 1000 positions and at 4-ply 10000 positions. If it takes 1/10 second to score each position, then a 1-ply search will need 1 second, 2-ply will need 10 seconds, 4-ply will need 1000 seconds and 8-ply will need over 3 years!

How deep your program searches will therefore depend on the speed it takes to carry out its evaluation, also allowing for the time taken to generate moves. A BASIC program can barely search beyond 2-ply for two main reasons:

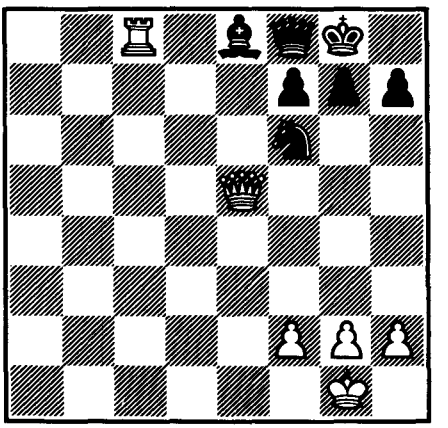
Firstly, it runs too slowly, and each EF will probably take a second or two to score.

Secondly, Atari BASIC will not permit 'recursion'; that is, a subroutine calling itself. It is obviously desirable that the program has only one move generator which is called as a subroutine at each level of search. In Atari BASIC, you would have to write a new move generator at each level. One way around this is described in Chapter 12.

These problems can be overcome by use of machine-code or, I believe, with a Pascal compiler. The high level language Pascal permits, indeed encourages, the use of recursion, and a compiler would give the necessary speed. However, I have no personal experience with Pascal, and in any case I know of no Pascal compiler for the Atari at the time of writing.

A search simply to a fixed depth can be very dangerous. Consider the chess position in **Diagram 3-3.**

Diagram 3-3



With a fixed search depth of 3-ply, the program will evaluate this sequence:

1. c8 x e8 f6 x e8
2. e5 x e8

as favourable (+3−5+3) and will carry it out, failing to notice the subsequent loss of the queen!

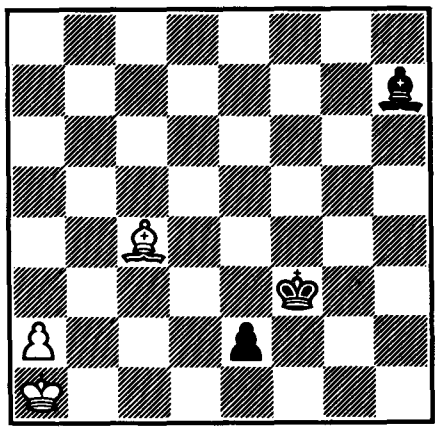
The English mathematician Turing was one of the first to point out that move-sequences must be searched in depth until no further captures are possible. This is known as a Turing-Dead position, and is widely used in the more sophisticated computer programs for such games as chess, draughts and Reversi/Othello.

The search deep into a line containing many alternative captures can be very time consuming, and it has been suggested that it may not be necessary if a sufficiently sophisticated EF is used. Thus in **Diagram 3-3,** the program will not capture the rook even with a fixed 3-ply search, since it sees that at the end of the move sequence the queen can be captured even though it has not YET been.

This illustrates the constant trade-off between the EF and searching in depth. It takes longer to score a large EF than it does to score several moves with a simple EF, but the more complicated EF may save much time searching at progressively deeper levels.

Another limitation of a fixed depth of search is the so-called *horizon effect.* A program may ignore the disastrous end of a move sequence, if it can interpose a series of pointless moves which push the disastrous culmination beyond its fixed depth of search; in other words, pushing the disaster over the 'horizon'. Consider **Diagram 3-4.**

Diagram 3-4



If the bishop does not take the pawn (which sacrifices the bishop), then the pawn will be promoted to a queen so the sequence is either c4xe2 ch

f3xe2, which scores +1−3=−2 for the program, or c4−b5 (say) e2−e1 (=Q) which scores +1−9=−8 for the program. Instead, if it has a fixed 3-ply search, it will play

1 c4−d5 ch f3−f2
2 d5−c4

which scores 0 for the program. The next opponent move, promotion of the pawn, has been pushed over the horizon by the check, which appears to lead to a better position than the exchange of the bishop for the pawn.

Even the best commercial chess programs can be prone to the 'horizon' effect.

Alpha-Beta

It takes a long time to search all the variations of a tree by the minimax algorithm, and any method which reduces the number of nodes to be scored will reduce that time significantly. Such a method is the *alpha-beta* algorithm which gives *identical results* to the Minimax search while searching fewer nodes. Consequently, Minimax should never be used in any program without also employing the alpha-beta modification. Discovered in 1959, the underlying principle of alpha-beta searching is that if a position is scored after an opponent move which is worse for the program than its previous best backed-up score, then there is no need to waste time searching the other nodes of that opponent move.

If you refer back to **Diagram 3-1,** the backed-up score for move A is −1. When considering move B, the first considered opponent move gives a score of −3, so that the backed-up score for move B must be, at best, −3 for the program. Since −3 is worse than the −1 already scored for A, all the other opponent responses to move B can be ignored, and move B can be dropped from consideration. This is called alpha pruning. On the other hand, none of the terminal nodes of move C are less than −1, so all the responses to move C will be evaluated in full. The value for move A is called the *alpha cut-off* value, since alpha pruning is carried out by reference to it. The cut-off value will be updated as better backed-up scores are found for the program.

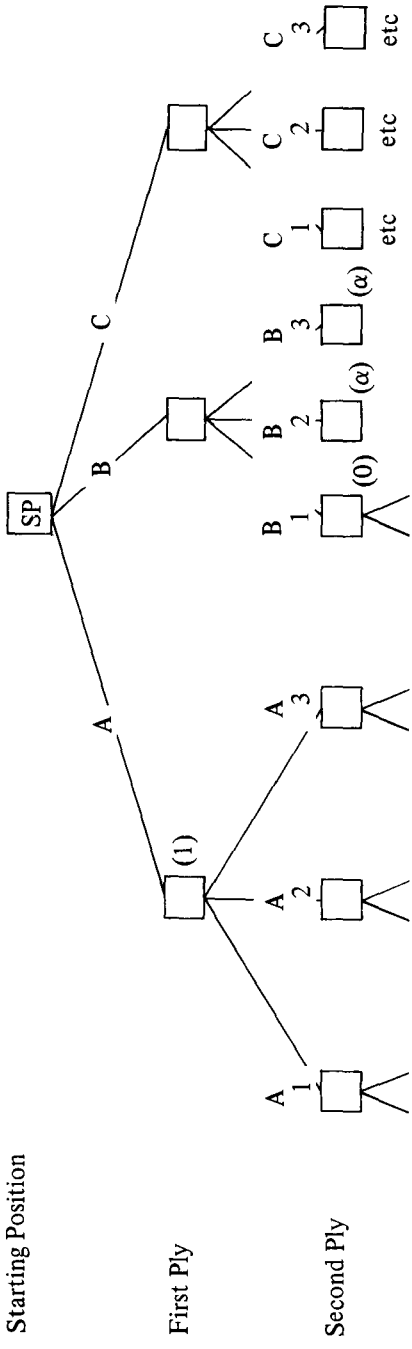
At deeper levels, with the opponent to move and considering program counter-replies, the alpha-beta algorithm works in reverse. This time, any terminal node which exceeds the move stored as worst will cause a cut-off of all the other nodes for that move. This is known as beta pruning, and has associated *beta cut-off* values.

In general, alpha pruning will occur at even ply levels and beta pruning at odd ply levels.

A further example is given in **Diagram 3-5**

The backed-up score for computer move A1 is 1. (Problem: Why?). The node A21 has a score of 2. This means that the backed-up score for

Diagram 3-5


Starting Position
First Ply
Second Ply
Third Ply
SP
A
B
C
(1)
(0)
(α)
(β)
etc

computer move A2 must be higher than 1, so the next 2 nodes are beta-pruned and move A2 is dropped. Similarly, move A31 scores higher than A1, so move A3 is dropped at once. This means that the backed-up value of move A is 1.

The backed-up score for move B1 is 0. This is worse than the backed-up score for move A, so moves B2 and B3 are alpha-pruned – no matter how good they are, the opponent will choose move B1.

Deep alpha-beta pruning

This section is difficult, but is unlikely to be needed by BASIC programmers since they will not search deep into the game tree anyway.

In 1973, Gillogly demonstrated that alpha-beta cut-off values could be used to prune the tree at any even number of levels below the cut-off value being considered. As an example, consider **Diagram 3-6.**

The backed-up value for move A is +3. The alpha cut-off is therefore 3. The backed-up value for move B is +6. This is better than move A for the program, so the alpha cut-off is now set to +6.

Looking 4-ply into the tree for move C, the program finds that move C111 scores +5. It means – since at this level the opponent is moving – that the best score the program can make from its move C11 at the 3rd ply is +5, which is lower than the alpha cut-off value already stored in alpha. Accordingly, moves C112 and C113 are deep alpha pruned (shown in the diagram as α α). Considering the next program move at 3-ply, the backed-up score for move C12 is +7 (minimising moves C121, C122 and C123) which exceeds the alpha cut-off. The alpha cut-off value is now set to 7.

Opponent move C131, in response to program move C13, scores +6, which is less than the new alpha cut-off value, and so moves C132 and C133 are alpha pruned in the usual way (shown in the diagram as α).

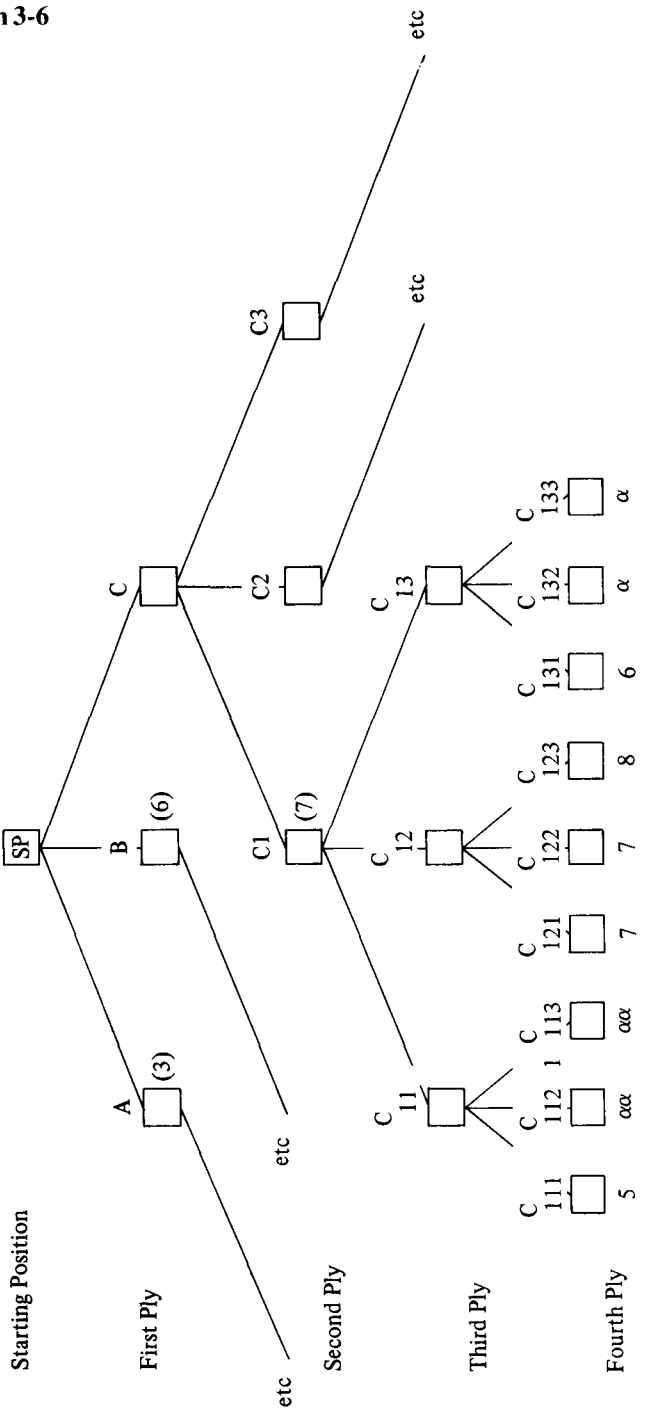
A similar method can be applied to deep beta pruning. It follows, then, that only two stores called ALPHA and BETA, which retain the updated alpha and beta cut-off values, are needed to alpha-beta prune the entire game tree. ALPHA and BETA are initially set to – infinity and + infinity respectively.

Nevertheless, you would be well advised at first to use more stores to avoid confusion. They should be set to – infinity for all program moves and + infinity for all moves by the opponent.

By constantly updating several stores holding the best moves for both sides at each ply level, it is possible to construct a best sequence of moves known as the *principle variation.*

For example, when evaluation is complete the best first ply move for the program would be piece X1,Y1 to U1,V1 held in store 1. The best second ply move for the opponent would be X2,Y2 to U2,V2 in store 2

Diagram 3-6


Starting Position
First Ply
Second Ply
Third Ply
Fourth Ply
SP
A
B
C
(3)
(6)
(7)
etc
C1
C2
C3
C 11
C 12
C 13
C 111
C 112
C 113
C 121
C 122
C 123
C 131
C 132
C 133
1
5
αα
αα
7
7
8
6
α
α
α

and so on until the end of the search is reached.

The first move of the principle variation (at 1-ply) is the move that the program will make. The second move (at 2-ply) is the best reply which it has found for its opponent. This move can be used to provide a hint to the opponent as to what he should do.

Although the alpha-beta mechanism is fairly easy to understand in principle, it can be horribly confusing to put into practice. Always, always test your Minimax program without the alpha-beta mechanism first, and then with the pruning routines added. You should get identical results every time, but remember to remove any random factors from the EF.

I remember sitting over a hot computer at one in the morning, trying to work out just what was going on. I had a bottle of the '79 Hainfelder Ordensgut Kabinett to hand, and, as I kept filling my glass, and sank slowly deeper under the table, I felt that I had at last come to a closer understanding of the alpha-beta mechanism.

In BASIC, we do not need to consider searching deeper than 2-ply, and the following routine will do the trick on a 8x8 chess board:

```
1000 REM MOVE OPPONENT PIECE
1010 AB=0:FOR I = 1 TO 8: FOR J = 1 TO 8
1020 IF AB = 1 THEN 1060
1030 IF A(I,J)>=0 THEN 1070
1040 (Move piece)
1050 (Gosub evaluate)
1060 (Restore piece)
1070 NEXT J: NEXT I
2000 REM  EVALUATION
2010 Q=0: (Evaluate)
2020 IF Q<QQ THEN QQ=Q : (store moves)
2030 IF Q< ALPHA THEN AB=1
2040 RETURN
```

where ALPHA is the best backed-up score (alpha cut-off) at the first ply, and QQ is the best response yet found for the opponent (worst for the computer). AB is the flag for alpha-beta pruning, operating when set at 1.

The program becomes more complicated if you are evaluating moves, instead of positions. See the move-storage routines for Warp Trog at the back of the book.

If you find this account of Minimax and alpha-beta pruning confusing – as you probably will – separate accounts can be found in D N L Levy's *Computers and Chess* (Batsford) and Birmingham and Kent's *Advances in Computer Chess* (Edinburgh University Press). It took me several readings of different accounts to understand the mechanism.

CHAPTER 4
Advanced Methods

In this chapter we shall discuss a number of more advanced algorithms and heuristics to enable the program to select its move. They will only be of interest to really dedicated programmers, and are not necessary for the beginner.

An algorithm is a procedure which is mathematically precise; its consequences can be exactly calculated. A heuristic is an empirical procedure, a 'rule-of-thumb' with no precise mathematical justification, but which experience shows will normally – but not always – give the result that we want.

Alpha-beta pruning and Minimax searching are both algorithms. With a knowledge of the game tree, we can always predict the end result.

The method works well, and is often used in amateur programs, but it is possible to improve it still further.

The improvement is effected by increasing the efficiency of alpha-beta pruning. We have seen that if the program has a backed-up score for move A of 1, then any opponent response to move B which scores less than 1 will lead to the rejection of move B; similarly any opponent response to move C which scores less than 1 will lead to the rejection of move C, and so on.

But what if all the opponent responses to move B exceed 1, leading to a new backed-up score for move B, which is in turn exceeded by all the opponent responses to move C? In that case, no alpha-beta pruning occurs at all. See **Diagram 4-1.**

The problem arises because the best move (C) is considered last.

If the program's EF is good enough, then it should already have guessed that move C would be strongest at its first ply of search, before searching deeper. Move B might also have been considered the second best.

If the program generated all its first moves at 1 ply before searching any deeper, it can then sort them into numerical order, putting the highest scoring move at the top of the list to be searched. This means that **Diagram 4-1** must now be rewritten as in **Diagram 4-2.**

The first backed-up score (for move C) is now 3. The first counter-move to move B is 2. This is less than the backed-up score for C, so both other nodes of B are pruned out.

Similarly, two of the three terminal nodes from move A are rejected

Diagram 4-1

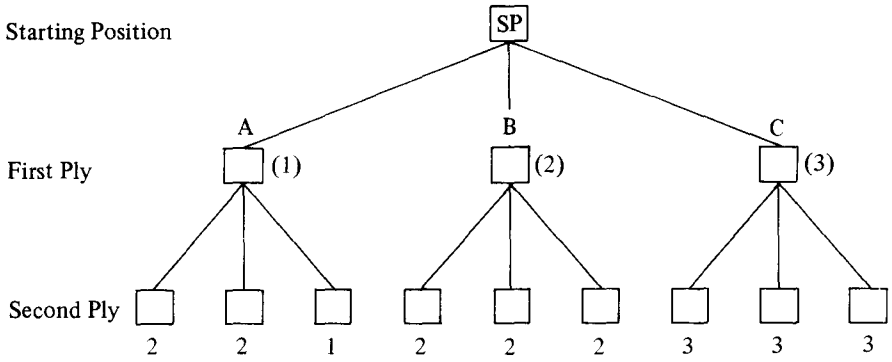
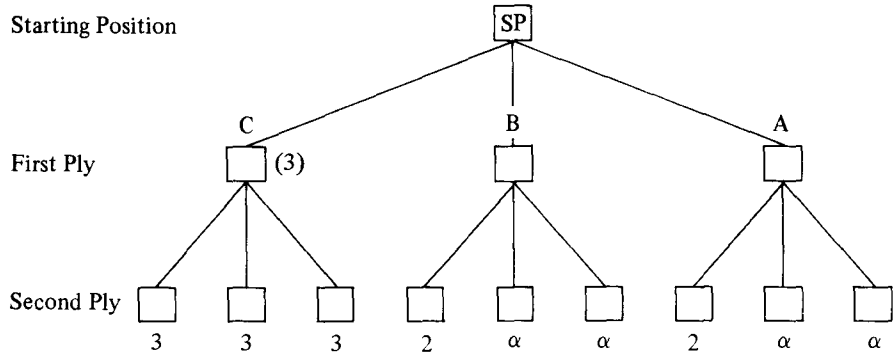


Diagram 4-2



after the first is evaluated as only 2.

In Diagram 4-1, the computer evaluated 9 positions at level 2. In Diagram 4-2 the program evaluated 3 positions at level 1 (prior to sorting) and 5 at level 2. This is a total of 8 nodes.

The saving in this instance is only one evaluation, but the example was a very simple one with only 3 program moves and 3 responses to each. In a more complicated game, such as chess with its average of 30 moves per side, the saving becomes enormous.

It is possible to show mathematically that proper ordering of the moves before using the alpha-beta algorithm can save (N–2*SQR(N)) of the total number (N) of terminal positions examined.

For example, if there are 10,000 terminal nodes, 9,800 nodes are not evaluated, only 200 are evaluated.

How well the program puts the moves into order depends on how effective its EF is. Again, there is a trade-off between the time taken to score a complex EF and the saving caused by more efficient pruning.

Similar considerations apply when the program considers its responses to the opponent's counter-moves. In principle, the opponent moves could be sorted before searching deeper. However, this increases the number of sorts needed, and the time taken to do so may exceed the savings of more efficient pruning. Only experimentation can determine its value for any program, but most good commercial chess programs – which face the severest test – change only the order of the program's moves, changing the order (if necessary) at each ply of search.

Another useful trick in this context, again used by many commercial chess programs, is to carry out a complicated positional evaluation for each move at the first ply of search (which may take a long time in computer terms, several seconds for 30 moves in chess) then search each subsequent ply level scoring the material count only. The material count can be evaluated very quickly indeed. There are problems with this approach, though. We shall discuss the method further in Chapter 11.

The program can sort its moves by any of the standard methods, moving into order all the moves and their associated scores. A simple bubble sort is sufficient in machine-code.

Combining deep search and sorting is rather time consuming in BASIC, and if you are sorting more than 10 moves, a 'fast-sort' (for example the Shell-Metzner method) is desirable. Such methods usually use a random factor in the sorting algorithm, so that equally-scored moves may be sorted on successive occasions into different orders. The result may cause you confusion while testing an alpha-beta algorithm!

This is how we might combine Minimax search with sorting in BASIC:

```
10 N=0:REM N=no of moves considered
20 (Find move) :N=N+1
30 (Move piece from location X,Y, to U,V)
40 (Evaluate position as Q)
50 X(N)=X:Y(N)=Y:U(N)=U:V(N)=V:Q(N)=Q
60 (Restore piece)
70 IF (Any more moves) THEN 20
80 FOR I = 1 TO N
90 (Sort X(N),Y(N),U(N),V(N),Q(N) into descending order of Q(N))
100 NEXT I
```

The above program generates all the computer's moves at the first ply, scores them and then sorts them into decreasing score order. The counter N keeps track of the number of moves.

As the next step, the computer would carry out each sorted move, from

square X(N),Y(N) to square U(N),V(N), generate all the opponent's responses, score the new positions, then replace its pieces from square U(N),V(N) back to square X(N),Y(N).

Suppose N = 0? Then the program has no legal moves left. In many games, such as draughts/checkers, it would mean that the program had lost the game. In chess, no legal moves means that the game is stalemated. In either case, routines should detect the N=0 condition and take the appropriate action.

Another interesting situation occurs when N = 1. It means that the program has only one legal move. The move can therefore be made at once, without need to search to any deeper level. Such drastic chopping off of all deep levels is known as the *chopper* mechanism. It appeared for the first time in commercial chess programs as late as 1980. Incorporate a routine in your program to test for N=1.

Iterative deepening

The whole process that I have described, finding and scoring the moves at the first ply, sorting, searching counter moves to the second ply, sorting and searching to the third ply and so on, is known as *iterative deepening.* (Iteration means increasing by 1, deepening refers to the ply level of the tree).

Because the program has its moves ordered so that its best-move-yet is always at the top of the list, the program can be interrupted at any time and the top move of the list will always be its best move to the level searched.

On the other hand, interrupting an un-sorted program may result in the best move being anywhere in the list. Commercial programs which permit you to halt their thinking and display the best move yet found – or which use timers which decrease to zero to interrupt the program – must inevitably use the method of iterative deepening.

By placing the best moves at the top of the list, iterative deepening also allows the program to carry out a selective search. It can, at any ply depth, simply take just the top few moves off the list – one of which is almost certain to be the best, however deep it searches – and then search these few moves deeper, if necessary selecting only the top few moves of the corresponding counter-moves.

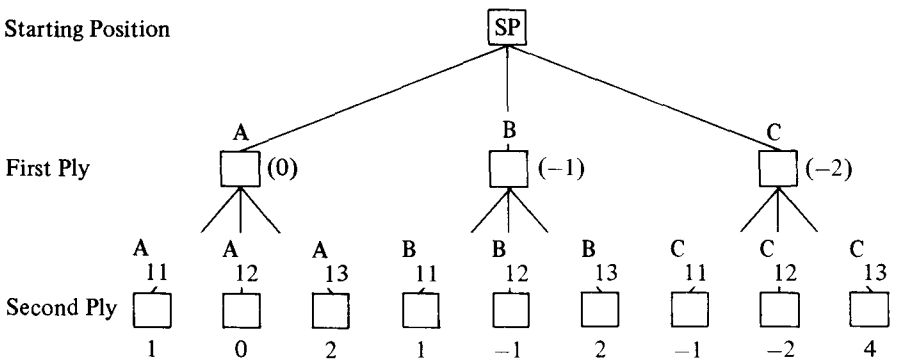
For example, if a chess program can find 30 initial moves, after searching to a depth of 3 ply, the 30 moves can be placed in order. The top six moves can be searched to a depth of 4 ply, and the top six opponent responses searched to a depth of 5 ply.

This method, or something similar, probably lies behind the claims for certain commercial chess programs that they use a selective search. It is very necessary to have a good EF, to ensure that the top six moves at each level contain the actual best move.

The alpha-beta pruning mechanism chops out part of the game tree at all levels down to the deepest. To make alpha-beta more effective at the penultimate level, it would be necessary to sort all the nodes at the last level into order: requiring generation of all those nodes, defeating the pruning objective.

It is reasonable to assume that any move which is best for the player at the deepest ply, in response to the second player's previous move, will also be good in response to any of the second player's other moves at that ply level. Such a best move should be stored and used before any other to try to refute all the second player's moves at the penultimate ply level. The move is called the *killer heuristic*, being empirically likely, but not mathematically certain, to refute the other player's moves. **Diagram 4-3** will illustrate the idea.

Diagram 4-3



The backed-up score for program move A is 0. The best opponent response was move A12. Move A12 is stored as the killer heuristic, and is the first tested in response to program move B. The method assumes that most, or all, of moves B11, B12, B13 and C11, C12 and C13 are the same as moves A11, A12 and A13, which is very likely in nearly all strategy games. However, they lead to different positions against different program moves A, B and C.

The killer heuristic is used to replace identical move B12, and is considered first in response to move B. The result (−1) is worse for the program than the backed-up score for move A, so moves B11 and B13 are at once alpha pruned.

Similarly moves C11 and C13 can be alpha pruned. Hence use of the killer heuristic has led to only five positions (3 to move A, 1 to move B and C) being evaluated. Ordinary alpha-beta pruning in the same position

would result in six positions being evaluated (3 to move A, 2 to move B, 1 to move C).

Problem: Which move (A,B or C) will the program make? (A)

The killer heuristic can be kept the same throughout the evaluation of the final level, or it can be updated as each series of moves is evaluated. Care must be taken that the killer heuristic actually exists at any given moment: if the killer is Q x P for most of the final ply level, then the sudden capture of the queen on the previous ply level makes the killer a nonsense. Hence, continually updating the killer is wiser.

The killer heuristic can also be used at other levels of the search tree, where the move held in the alpha or beta stores serves as the killer. One particular application where the killer heuristic is especially effective is in problem solving of the 'mate-in-four' kind that can be found in chess.

An example of the killer heuristic in action can be seen in Warp Trog at the end of the book.

I have already mentioned that the alpha-beta pruning mechanism can be operated by two stores, set initially at + infinity and at − infinity (decimal 32767 or $7FFF in signed two's complement two-byte machine code). A somewhat esoteric advance is the use of the *alpha-beta window*, where the alpha and beta stores are set initially much closer to zero than to + or − infinity.

The rationale is that the program assumes that no truly good, or spectacularly bad, sequence exists for either side at any given moment. By narrowing the alpha-beta pruning range, superficially very good or very bad positions – which we assume will not be so good or so bad on closer inspection – can be pruned out at once.

Obviously, this requires careful setting of the alpha and beta stores to work well. A special case occurs when we assume that the program (or its opponent) can always find at least one move that will make the position for the player moving better than it was before the move. In this case, the technique is known as *razoring*, and its original proponents claimed a ten-fold acceleration in search time, relative to ordinary ordered alpha-beta pruning.

If no moves, or very few moves, are found after using an alpha-beta window, then the alpha and beta stores are reset to + and − infinity and the moves are re-examined, resulting in a waste of time.

Hard pruning

The above techniques are all that are needed to make a very strong move at any game of strategy, and some of the world's strongest chess computers use no other pruning methods.

However, another common method of tree pruning exists. Known simply as *hard pruning*, it relies solely on cutting out all moves which fail

to exceed a certain score which the programmer has preset. Thus, in a game of chess, any move which gives a backed-up score of less than (say) −2 (loss of two pawns) from the original position is simply chopped out at the lowest level of search where the low score is encountered.

The method, although crude, can be surprisingly successful provided that the evaluation function is a good one. The early Chess Challenger models (not the present ones) used hard pruning and played generally rather sensibly, although they would rarely sacrifice a queen even to avoid checkmate, since moves involving the queen sacrifice were pruned out early on.

The big problem with this approach occurs if all the projected moves score so badly that they are pruned out. It is necessary to test whether a sufficient number of moves have been generated; if not, the pruning score limit (set at −2 in the example) must be lowered to enable extra moves to be considered.

Hard pruning need not be implemented at all levels; pruning can be started only at deeper levels and can also be done on the opponent's responses.

Another complicated trick is to save programming time by storing the best sequence of moves found. Then, if the opponent on his turn plays the next move in the sequence, the program already has a good first move to hand to start from with its likely sequel, although a new deep search may cause the sequence to be modified.

A useful addition to the EF is a heuristic which tells whether a program is ahead in material or not. If the score from the heuristic is added to the EF, then the program will tend to make exchanges if it is winning, or avoid exchanges if it is losing.

The best known such heuristic is due (I think) to the American checkers programmer A Samuel, who formulated the heuristic

Q = Q + (Stronger side's material)/(Weaker side's material) * (Program material − opponent material)

where Q is the evaluation score.

Finally, it is often useful to evaluate the starting position before any move is considered. It will give important information about the state of play, e.g., whether the endgame has been reached, and other applications have already been described.

If you have waded your way this far through the preceding three chapters, you are probably feeling rather like a character from Tennyson who has been hurried from sport to sport without finding much to smile at.

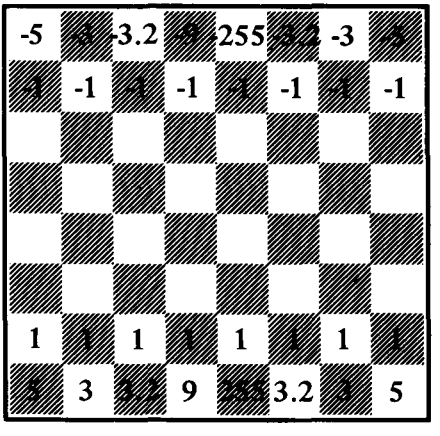
For light relief, turn now to Chapter 7 where two 1-ply games written in BASIC are described. Type them in, but at this stage concentrate only on what the EF is achieving.

CHAPTER 5
Set up your Game Board

One of the most important features of any game of strategy is setting up the initial position. It requires more thought than is often given.

Virtually all games of strategy are played on boards, and this must be examined first. In the examples given earlier in this book, a two-dimensional array A(X,Y) was used to represent the board. A complete chess board could therefore be represented as shown in **Diagram 5-1,** where all the computer's pieces are assigned positive values, and the opponent's are negative. The values represent the nominal value of the pieces.

Diagram 5-1



Note that it is necessary to give the bishop a slightly higher value than the knight to distinguish both pieces.

All the remaining squares are set to zero. The standard Atari BASIC does not clear reserved array space when the program is RUN, so the first step needed is a program line to do this job. Otherwise , your free squares may contain odd numbers like −0.210079E−10 (really!)

A BASIC program to set up and display the chess board would be **Program 5-1.**

```
5 REM PROGRAM 5-1
10 DIM A(8,8)
20 FOR I=1 TO 8:FOR J=1 TO 8
30 A(I,J)=0
40 NEXT J:NEXT I
50 FOR I=1 TO 8
60 READ MYPIECE
70 A(I,1)=MYPIECE:A(I,8)=-A(I,1)
80 A(I,2)=+1:REM PLACE PAWN
90 A(I,7)=-1:NEXT I
100 REM ** DISPLAY BOARD **
105 GRAPHICS 0:POKE 752,1
110 FOR I=1 TO 8:FOR J=1 TO 8
120 POSITION I*4,(9-J)*2:PRINT A(I,J)
130 NEXT J:NEXT I
140 POSITION 0,20
150 DATA 5,3,3.2,9,255,3.2,3,5
```

In practice, for reasons we shall see in the next chapter, it is often desirable to add a rim all the way around the board. The rim is a series of squares whose array values must be set to a value distinguishable from the rest of the board, and from any piece. In the chess example, we could give the rim a value of 7. Change line 10 in **Program 5-1** to read:

```
10 DIM A(9,9)
```

and change line 110 to read

```
110 FOR I = 0 TO 9: FOR J = 0 TO 9
```

then add lines

```
95 FOR I=0 TO 9
96 A(I,0)=7: A(0,I)=7: A(I,9)=7: A(9,I)=7
97 NEXT I
```

and re-RUN the program. The board now has a solid rim.

In games of chess, a second rim is often needed and this means that in BASIC the whole board array would inconveniently need to be moved up. However, few people will write chess programs in BASIC, so this need not concern the writer of BASIC strategy games.

It is sometimes desirable to have two separate boards representing the position, but inverted with respect to each other. It means that all moves by both sides appear, to the program, to be going the same way which simplifies the move generator. For the same reason, the rim may change sign as the program searches each successive ply; that is, it will be +7 at

each ply level where the program moves (odd ply depths) and −7 at each ply level where the oponent moves (even ply depths). Such modifications are sometimes convenient, sometimes not, depending on the game.

Atari BASIC, like many BASICS, uses six consecutive bytes in memory to hold a single floating point number as its BCD representation.

It does not permit integer-only BASIC to be used, which would require only two bytes per number. Thus the array A(8,8) requires no fewer than 8*8*6 = 384 bytes just to hold a few simple integers such as 1, 0 and 9.

We can reduce the memory requirement by putting the board into a single 'vector' stored in a string. Atari BASIC requires only one byte to store each element of a string. The expression A$(64) requires just 64 bytes. Unfortunately, Atari BASIC does not support Microsoft-style multidimensional string arrays (eg A$(8,8)). This is probably Atari BASIC's single worst failing. However, we can store the board as a single vector in a one-dimensional string. There seems to be no limit (except memory) up to 32767 as to the size of a string array, which will accommodate any reasonable game board.

The string vector is constructed by multiplying the vertical coordinate less one of any square on the board by the depth of the board, and adding the horizontal coordinate.

In general, the string vector value (V) of any square X,Y on an 8x8 board is

V = X + (Y−1)*8

Thus, square 2,3 of an 8*8 chess board has a string location of 2+(3−1)* 8 = 18, while square 3,2 has a string location of 11.

If you are incorporating a rim into the board, you should allow for the extra width of the board. A double-rimmed chess square will occupy 12* 12 locations.

We can now write

V = (X+2) + (Y+1)*12

Thus square 2,3 will now have a string location of 2+2 + (3+1)*12 = 52.

Problem: On an ordinary chess board, what are the string locations of squares A,1 and H,8? (1 and 64)

What are the locations of the same squares on a double-rimmed board? (27 and 118)

To fill the string, it is necessary to work out the string locations of every piece, and also of every rim square, and fill the rest of the string with zeros.

Example: On a 3*3 square with 1 rim, there are two pieces, value +1 and +2, at locations 1,3 and 2,2. The string vector will be

"77777 70007 70207 71007 77777"

(the spaces are put in for clarification, and must not appear in the actual string). Note that the piece of value +1 has a string vector value of 17, and it appears in the 17th position of the string as we would expect. The

7s are rim values.

The method has certain limitations. For one thing, how would you put in an opponent piece, e.g., -5? It is possible to do this with string-slicing methods (all positive numbers must be put in as, e.g., +5), but is rather tedious. Again, no number above 9 can be used without further tedious spadework. If you wanted to use the number −100, then all other numbers would have to be expressed as +001, −001, +010 as required.

However, the method has certain advantages for machine code programmers. The array can be filled with 1-byte figures in two's complement notation. Thus a positive number up to 7F(hex) can be stored as one byte, while the opponent pieces can be stored as their two's complement form. For example, an opponent piece of value 3 can be stored as (256–3) = FD(hex).

Using this method, the highest value that can be stored for any piece is 7F (127 dec). This is still acceptable as a value for the indispensable king in chess, since it exceeds the combined sum of all other pieces (including 9 queens) which comes to 103(dec).

It still leaves us with the problem of defining the exact value of the pieces. Earlier, we gave a knight a value of 3 units, while a bishop was rated at 3.2. Such values cannot be accommodated in one-byte numbers, and in any case setting a pawn at a value of +1 does not allow any scope for the subtleties of positional evaluation, which is normally worth less than a pawn.

One solution is to assign each piece a two-byte signed value, with the most significant byte having the nominal value of the piece. For example, a pawn would be 0100, and an enemy pawn FF00. Consequently the string vector for the board must be assigned two bytes per square of the board and rim.

The method of storing pieces on a board as their nominal values works satisfactorily and is convenient in BASIC. However, it should be clear from the previous paragraphs that it is somewhat less convenient in machine code.

A better method for the machine code programmer – which also works in BASIC – is to assign each piece a 'token' value. Thus, in chess, each pawn would be assigned the token value of 1, each knight a token value of 2, each bishop a token value of 3, each rook a token value of 4 and the queen and king token values of 5 and 6.

A table of look up values is used to attribute the piece values to the piece tokens. For example, the program searches the board, finds a piece token of 4 at square 1,1 and looks up the 4th value in the table, which has a value of 5. The program then knows that the piece at square 1,1 is a rook (token 4) with a value of 5 units.

We still have to assign token values to the opposing pieces. They can be given negative values corresponding to the positive values of the program pieces, or they can be given completely separate token values

of their own.

Program 5-2 given below initialises a chess value look up table in the array B(6), and allows you to place token values of chess pieces on the board at location X,Y in the form X,Y,TOKEN (return). The program then searches the board until it finds the token and prints out the token with its location and the piece value.

```
5 REM PROGRAM 5-2
10 DIM A(8,8),B(6)
20 FOR I=0 TO 6
30 READ C:B(I)=C:REM FILL LOOK UP TABLE
40 NEXT I
50 DATA 0,1,3,3,2,5,9,255
55 REM DATA CONTAINS PIECE VALUES
56 GRAPHICS 0
60 FOR I=1 TO 8:FOR J=1 TO 8:A(I,J)=0:NEXT
   J:NEXT I:REM CLEAR BOARD
70 PRINT "ENTER X,Y, TOKEN"
80 INPUT X,Y,TOKEN
90 IF X>8 OR Y>8 OR TOKEN>6 THEN 70
100 A(X,Y)=TOKEN
110 FOR I=1 TO 8:FOR J=1 TO 8
120 Z=A(I,J):IF Z<>0 THEN VALUE=B(Z):PRINT
    "TOKEN = ";TOKEN;" LOCATION ";X;" ";Y
    ;" PIECE VALUE = ";VALUE
130 NEXT J:NEXT I
140 A(X,Y)=0
150 GOTO 70
```

So far, we have only discussed two-dimensional games boards, and their one-dimensional representations. There is, of course, no reason why a three-dimensional board should not be used, as in 3-D noughts and crosses, or in the 3-dimensional chess played by Mr Spock in Star Trek (TM).

In theory, you could play any game of strategy in as many dimensions as you like. However, only computers would be likely to play 10-dimensional chess even remotely well; human players would have trouble visualising the moves!

Screen display

The Atari computers offer excellent sound and graphics facilities which can be used to enhance any program.

Graphics modes 0, 1 and 2 are probably the most useful in this context,

allowing characters to be printed on the screen, whereas levels 4 to 8 (and 9 to 11 if you have the GTIA chip) are map modes which are mostly used to plot individual pixels. In addition, ANTIC – the screen display chip – offers a further three character modes which are best accessed through a customised Display List.

Graphics mode 0 offers a screen display of 40 by 24 characters in two colours, while modes 1 and 2 allow respectively displays of 20 by 20 and 20 by 10 characters in up to four colours.

A full account of the use of Atari graphics and customised character sets is beyond the scope of this book; an excellent description can be found in *Your Atari Computer* by Lon Poole (Osborne/McGraw Hill).

The Atari reference manuals supplied with the 400/800 computers lamentably make no mention of user-selectable character sets. The original 1K character set – 128 characters – supplied with the Atari BASIC cartridge is in ROM, and cannot be modified, but it is possible to copy the set into free memory above your program where they can be altered to suit the programmer's whim.

Each character consists of 8 bytes, and a useful character set for chess pieces is given in **Table 5-1.**

Table 5-1

PIECE	DATA FOR CHARACTER
Pawn	0,0,16,56,56,16,124,0
Knight	0,16,56,120,24,56,124,0
Bishop	0,16,40,68,108,56,124,0
Rook	0,84,124,56,56,124,124,0
Queen	0,84,40,16,108,124,124,0
King	0,16,56,16,56,124,124,0

The use of this character set is illustrated in program 8-1

Character sets must be copied onto 1K boundaries of free memory if in graphics mode 0. Because of the increased memory needed to store four colours in modes 1 and 2, only half of the character set can be copied, and it must start at a ½K boundary.

It takes several seconds to copy 1024 or 512 bytes from ROM into free memory, and the machine code program, **Program 5-3**, will do the trick much quicker.

```
5 REM PROGRAM 5-3
10 DIM E$(37)
20 F=(PEEK(106)-8)*256:G=4:REM G = NO. OF
   PAGES TO COPY. 1 PAGE = 256 BYTES
```

```
30 FOR I=1 TO 36
40 READ A:POKE ADR(E$)+I,A
50 NEXT I
60 Q=USR(ADR(E$)+1,224*256,F,G)
70 DATA 104,104,133,205,104,133,204,104,133,
   207,104,133,206,104,104,133,208,166,
   208,160,0
80 DATA 177,204,145,206,200,208,249,230,205,
   230,207,202,208,240,96
```

When in position, the new character set can be called with

POKE 756, F/256

after each graphics statement. Individual characters can be given one of up to four colours by a COLOR statement, after first setting the four colour registers with the SETCOLOR command. The relationship between the COLOR command for the piece, and which COLOR register it uses, is complex and can be deduced from the Atari BASIC manual or from Table 11-4 of *Your Atari Computer.*

The Atari computers also offer up to four player-missile sprites. Although highly spectacular in arcade shoot-outs, I have never found a serious use for these in strategy games. However the American book *De Re Atari* suggests that they could be used to add extra, independent colour to the original screen. Another application can be seen in Program 5-6, where one of the four players serves as a cursor to move pieces on the board.

Sound statements should also be used to indicate when the program has made its move and to accept or reject (different sound) the user's input. A simple beep can be programmed as in **Program 5-4**:

```
5 REM PROGRAM 5-4
10 FREQ=80:DELAY=60
20 SOUND 0,FREQ,10,6
30 FOR I=1 TO DELAY:NEXT I
40 SOUND 0,0,0,0
```

Experiment by changing the sound frequency and the delay.

The program must accept the user's input, and by far the commonest means of doing this will be through the keyboard.

Many American software writers exclusively use the Atari joystick as a means of accepting user input, on the grounds that it reduces the chance of the user making a mistake.

As an example, Atari's own chess program in ROM requires use of the joystick to move each piece. This is fine for the complete novice who does not understand algebraic chess notation, but excruciatingly tedious for someone who does. And, after a time, one would expect the user to become more proficient, not less, in chess notation. For years, I made do with Descriptive Chess Notation (eg, P–K4, P–Q4) for recording chess games, but within four weeks of buying my first chess computer, I had become highly proficient at keying in moves with computer-acceptable Algebraic Notation.

As an alternative to the joystick, the user should type in moves at the keyboard. Unwanted keys can be masked out. A suitable routine, which gets a character from the keyboard and prints it at location 10,10, is given in **Program 5-5.**

```
5 REM PROGRAM 5-5
10 GRAPHICS 1:POKE 752,1
20 CLOSE #1:OPEN #1,4,0,"K:"
30 GET #1,LETTER
40 POKE 694,0:REM DISABLE INVERSE KEY
50 POKE 702,64:REM DISABLE LOWER CASE KEY
60 POSITION 10,10:PRINT #6;CHR$(LETTER)
70 GOTO 20
```

At this stage, the program can also conveniently do any necessary alphanumeric conversion; that is, if the move A2 is typed in, the letter A can be converted to the number 1 by the routine

NUMBER = LETTER – 64

(LETTER is actually the ASCII value of the typed character).

The Atari computer has a useful error-trapping mechanism, called TRAP, which should be used to detect any likely input errors. TRAP must be cleared before each re-use. An example would be:

```
10 TRAP 1000
20 (INPUT ROUTINE)
30 (REST OF PROGRAM)
1000 TRAP 40000: PRINT "ILLEGAL MOVE": GOTO 10
```

A subroutine also exists to disable the Break key of the computer. The subroutine is:

```
1000 I=PEEK (128)–128
1010 IF I<0 THEN RETURN
1020 POKE 16,I
1030 POKE 53774,I
1040 RETURN
```

The subroutine should be called after every graphics statement.

We now come to the vexed question of error-trapping versus tamper-proofing. If the program expects an input of a number, then clearly the program should guard itself against receiving a letter instead. Such a mistake would be natural. The user should always be protected from such errors.

On the other hand, is there really any reason to mask the BREAK and SYSTEM RESET keys? I don't believe that anyone could touch these keys accidentally – I know I never have – and therefore only deliberate tampering will operate them. Whether accidental or deliberate, the resultant crash will certainly deter the user from doing it again. So, I do not see any reason to make programs tamper-proof. In this view, I am at variance with most American authors whose views are possibly coloured by their litigious climate ('The program crashed when I meddled? Let's sue the programmer for mental distress!')

This is a matter of philosophy, which the program writer must decide for himself.

The last program in this section, **Program 5-6,** illustrates how player–missile graphics can be used to move pieces on a board (the reader is assumed to understand the principles of player–missile routines; if in doubt, see *Your Atari Computer* or *De Re Atari*).

I have included this program since I recognise that not everyone will share my views about using the keyboard instead of the joystick. Plug a joystick into port 1, then move it about the board. Press the trigger to pick up any of the numbered pieces, move the joystick again, and press the trigger to release the piece. Only legal moves (i.e., moves to vacant squares) will be accepted.

```
5 REM PROGRAM 5-6
10 DIM A(8,8),UP$(21),DOWN$(21)
20 FOR I=0 TO 20:READ B:POKE ADR(UP$)
   +I,B:NEXT I
30 FOR I=0 TO 20:READ B:POKE ADR(DOWN$)
   +I,B:NEXT I
40 DATA 104,104,133,204,104,133,203,160,1,
   177,203,136,145,203,200,200,192,11,208
   ,245,96
50 DATA 104,104,133,204,104,133,203,160,10,
   177,203,200,145,203,136,136,192,255,2
   08,245,96
60 FOR I=1 TO 8:FOR J=1 TO 8
70 A(I,J)=0
80 NEXT J:A(I,5)=I:NEXT I
90 A=PEEK(106)-8
```

```
100 POKE 106,A:GRAPHICS 2
110 POKE 54279,A
120 PMBASE=256*A
130 FOR I=PMBASE+512 TO PMBASE+639:POKE
    I,0:NEXT I
140 POKE 559,46:POKE 53277,2
150 POKE 53256,0:REM PLAYER SIZE
160 POKE 704,200:REM PLAYER COLOUR
170 POKE 623,1:REM PLAYER PRIORITY
    OVER PLAYFIELD
180 X=100:Y=80:HORIZ=53248:REM HORIZONTAL
    POSITION
190 FOR I=0 TO 7:READ B:POKE PMBASE+512+I+Y,
    B:NEXT I:REM PLAYER SHAPE
200 DATA 255,129,129,129,129,129,129,255
210 FOR I=1 TO 8:FOR J=1 TO 8
220 IF A(I,J)=0 THEN COLOR 176:PLOT I+5,
    9-J:GOTO 240
230 COLOR A(I,J)+48:PLOT I+5,9-J
240 NEXT J:COLOR 144+I:PLOT I+5,9:COLOR
    144+I:PLOT 5,9-I:NEXT I
250 GOSUB 430
260 A1=INT((X-96)/8+0.5)+1:A2=9-(INT((Y-24)
    /8+0.5)+1)
270 PIECE=A(A1,A2):IF PIECE=0 THEN GOSUB
    500:GOTO 250
280 PRINT "MOVE FROM ";A1;"  ";A2
290 SOUND 0,40,10,6
300 FOR K=1 TO 60:NEXT K:REM TRIGGER DELAY
310 SOUND 0,0,0,0
320 GOSUB 430
330 B1=INT((X-96)/8+0.5)+1:B2=9-(INT((Y-24)
    /8+0.5)+1)
340 PIECE2=A(B1,B2):IF PIECE2>0 THEN GOSUB
    500:GOTO 250
350 PRINT "TO ";B1;"  ";B2
360 A(A1,A2)=0:A(B1,B2)=PIECE
370 COLOR A(B1,B2)+48:PLOT 5+B1,9-B2
380 COLOR 176:PLOT 5+A1,9-A2
390 SOUND 0,50,10,8
400 FOR K=1 TO 60:NEXT K:REM TRIGGER DELAY
410 SOUND 0,0,0,0
420 GOTO 250
430 IF STRIG(0)=0 THEN RETURN
```

```
440 IF STICK(0)=14 AND Y>24 THEN A=USR(ADR
    (UP$),PMBASE+511+Y):Y=Y-1
450 IF STICK(0)=13 AND Y<80 THEN A=USR(ADR
    (DOWN$),PMBASE+511+Y):Y=Y+1
460 IF STICK(0)=11 AND X>96 THEN X=X-1
470 IF STICK(0)=7 AND X<152 THEN X=X+1
480 POKE HORIZ,X
490 GOTO 430
500 SOUND 0,120,10,6:POSITION 7,0:PRINT #6;
    "ILLEGAL":FOR K=1 TO 100:NEXT K
510 SOUND 0,0,0,0:POSITION 7,0:PRINT #6;"
    ":RETURN
```

CHAPTER 6
The Move Generator

Having established the board, both internally and on the screen, the program can now move its pieces. But before moving them it has to find them!

One method to find the program's piece, when the time comes to move it, is to search through the board until a square is found with a positive value, denoting that a program piece is located there. The opponent's pieces will have negative scores, and can be similarly located.

This method works well, but is a little slow. Whether a chess program has all of its chess pieces intact, or just a king and a pawn, it must still search through a whole chessboard (64 squares) to find its pieces. The problem is exacerbated on a larger board. My most recent strategy game Convoy Battle – soon to be released for the Atari – moves up to five ships on a 30*20 board. That's 600 squares to examine!

The solution is to keep the locations of all the pieces separately in their own tables, one for each side. These tables are known unsurprisingly as Piece Tables.

Instead of searching through all the squares of the game board, the program just has to search through its piece tables for the locations of the pieces of each side. The piece table does not normally – except in very simple games – replace holding the pieces on the game board; it is a supplement.

In many commercial chess programs, all the major pieces are stored in a piece table, but the pawns are not. It is possible by this means to find attacks by one piece on another with great rapidity. This is particularly important in chess where attacks on the king or queen by either side must be quickly discovered.

An example of the operation of a piece table in BASIC follows:

Program 6-1

```
5 REM PROGRAM 6-1
10 DIM A(8,8),A$(64)
20 FOR I=1 TO 8:FOR J=1 TO 8
30 A(I,J)=0
40 NEXT J:NEXT I
50 A(2,3)=5:A(5,4)=3
```

```
60 A(7,7)=-5:A(8,8)=-3
70 A$="23547788"
80 FOR I=1 TO 2:REM COMP PIECE
90 X=VAL(A$(I*2-1,I*2-1)):Y=VAL(A$(I*2,I*2))
100 PRINT "PIECE NO ";I,X,Y,A(X,Y)
110 NEXT I
120 FOR I=1 TO 2:REM OPPONENT PIECE
130 X=VAL(A$(I*2+3,I*2+3)):Y=VAL(A$
    (I*2+4,I*2+4))
140 PRINT "PIECE NO ";I,X,Y,A(X,Y)
150 NEXT I
```

Note that the program uses one piece table to hold both sides' pieces. Two piece tables could also have been used.

The moves made by the located piece depend on what sort of piece it is. In any case, only one common memory store should be used to hold the current location of the piece and each of its new locations. In a two-dimensional game, the memory store would have four locations, two to describe the starting point and two to hold each new move. Thus, in a chess game, the queen would be initially located at X,Y. Each generated legal move would be to a new location at U,V, both U and V being constantly updated. When the next piece was considered, X,Y would be updated to the new piece's location and U and V repeatedly modified.

All moves must be tested for legality. In particular, the move should not stray off the board. On a game board which measures N by N squares, the requirement that the piece stays on the board is

IF (N+1−U)*U > 0 AND (N+1−V)*V > 0 THEN (accept move)

Oddly enough, the simpler and equivalent expression

IF U > 0 AND U < N+1 AND V > 0 AND V < N+1 THEN (accept move)

actually evaluates faster, particularly when comparing machine code versions.

Faster still is the test:

IF A(U,V) <> 7 THEN (accept move)

Here we see the value of the board rim mentioned in the previous chapter, where the rim was set as 7.

You can visualise this more clearly by picturing a rook moving down a file. Either it can cautiously stop at each square and test whether the square is on the board, or it can thunder down until it bounces off the rim.

In chess, knights have awkward habit of hopping over the board rim. For this reason, you will need a double rim around the board – it cannot leap over both.

It is often helpful to make the rim change sign, according to which side is moving (see previous chapter).

The legality of other moves depends on the rules of the game. Few strategy games, however, will allow the program to move onto its own pieces, and a test to see if the target square is occupied is usually necessary. Exceptions to this rule include Ludo and Backgammon.

The general procedure to move a piece from square A(X,Y) temporarily to square A(U,V), to evaluate the position arising, can be seen in **Program 6-2:**

Program 6-2

```
10 C = A(U,V): A(U,V) = A(X,Y): A(X,Y) = 0
20 (Evaluate position)
30 A(X,Y) = A(U,V): A(U,V) =C
```

When the best move is found, then move the piece permanently with line 10 alone.

It will also be necessary when making moves to update the piece table (although this can often conveniently be left until the best move has been decided and made). See **Program 6-3:**

Program 6-3

```
5 REM PROGRAM 6-3
10 DIM A$(20),A(9,9)
20 A$="56....."
30 A(5,6)=1:I=1:REM PIECE NUMBER
40 X=VAL(A$(2*I-1,2*I-1)):Y=VAL(A$(2*I,2*I))
50 U=X+1:V=Y+1
60 C=A(U,V):A(U,V)=A(X,Y):A(X,Y)=0
70 PRINT "PIECE TABLE WAS: ";A$
80 PRINT "PIECE MOVED FORWARD BY ONE
   SQUARE DOWNWARDS AND SIDEWAYS"
90 A$(1,2)=STR$(10*U+V)
100 PRINT "PIECE TABLE IS NOW: ";A$
```

Problem: RUN Program 6-3 substituting actual values for P and Q (between 1 and 8).

Note carefully line 70. This is a convenient way in Atari BASIC of modifying the string holding the piece table. If you were programming in machine code, it would be simply a question of updating the appropriate memory locations of the piece table.

It will be instructive to see how we actually program a move generator in BASIC. I have selected the game of Hexapawn as my example, since it is a simple game requiring a 1-ply search by the program. Hexapawn is played on a 3x3 board, with three computer pawns on the top rank and three of the user's pawns on the bottom.

The objective of the game is for both sides to advance their pawns until they reach the other side of the board. The first player to do this has won. The moves are simple pawn moves: one square forward at a time if the path is unobstructed by any other piece, capturing diagonally forwards if an opposing pawn is on the appropriate square.

In the original Hexapawn, one side loses if it cannot make a legal move and the first side to move should invariably lose. I have considered positions where no legal move exists, for the side whose turn it is to move, as stalemate.

Program 6-4

```
100 REM ** HEXAPAWN **
110 DIM A(3,3),A$(6)
120 FOR I=1 TO 3:FOR J=1 TO 3
130 A(I,J)=0
140 NEXT J:NEXT I
150 EVALUATE=550:MOVE=470:SCREEN=660
160 A(1,3)=1:A(2,3)=1:A(3,3)=1
170 A(1,1)=-1:A(2,1)=-1:A(3,1)=-1
180 GRAPHICS 1:POKE 752,1:POSITION 6,0:PRINT
    #6;"HEXAPAWN":GOSUB SCREEN
190 PRINT "DO YOU WANT TO GO FIRST (Y/N)?"
200 INPUT A$:IF A$="Y" THEN 330
210 IF A$="N" THEN 230
220 GOTO 200
230 QQ=-100
240 FOR I=1 TO 3:FOR J=1 TO 3
250 IF A(I,J)=1 THEN GOSUB MOVE
260 NEXT J:NEXT I
270 IF QQ=-100 THEN POSITION 1,15:PRINT #6;"
    STALEMATE!":GOTO 740:REM NO LEGAL MOVES
280 A(U1,V1)=A(X1,Y1):A(X1,Y1)=0:REM
    MOVE PIECE ON BOARD
290 GOSUB SCREEN
300 IF V1=1 THEN POSITION 4,15:PRINT #6;"
    I WIN!":GOTO 740
310 POSITION 0,15
320 REM ** YOUR MOVE **
```

```
330 TRAP 760:PRINT "FROM ":SOUND 0,80,10,6:
    FOR I=1 TO 60:NEXT I:SOUND 0,0,0,0
340 INPUT X,Y:IF X<1 OR X>3 OR Y<1 OR Y>3
    THEN 330
350 PRINT "TO ":INPUT U,V:IF U<1 OR U>3 OR
    V<1 OR V>3 THEN 350
360 IF A(X,Y)<>-1 THEN 330
370 IF ABS(U-X)>1 THEN 330
380 IF V-Y<>1 THEN 330
390 IF U=X AND A(U,V)<>0 THEN 330
400 IF ABS(U-X)=1 AND A(U,V)<>1 THEN 330
410 IF A(U,V)=-1 THEN 330
420 A(U,V)=A(X,Y):A(X,Y)=0
430 GOSUB SCREEN
440 IF V=3 THEN POSITION 4,15:PRINT #6:"
    YOU WIN!":GOTO 740
450 GOTO 230
460 REM ** FIND MOVES **
470 X=I:Y=J
480 U=I:V=J-1:IF A(U,V)=0 THEN GOSUB
    EVALUATE
490 IF I-1<1 THEN 520
500 U=I-1:V=J-1:IF A(U,V)=-1 THEN GOSUB
    EVALUATE
510 IF I+1>3 THEN 530
520 U=I+1:V=J-1:IF A(U,V)=-1 THEN GOSUB
    EVALUATE
530 RETURN
540 REM ** EVALUATE **
550 Q=0:C=A(U,V):A(U,V)=A(X,Y):A(X,Y)=0:REM
    TEMPORARILY MOVE PIECE
560 FOR M=1 TO 3:FOR N=1 TO 3
570 IF A(M,N)=1 THEN Q=Q+(4-N)*(4-N):IF N=1
    THEN Q=Q+20:REM ADVANCE PAWN
580 IF A(M,N)=-1 THEN Q=Q-N*N
590 Q=Q+A(M,N):REM MATERIAL COUNT
600 Q=Q+RND(1):REM RANDOM FACTOR
610 IF Q>QQ THEN QQ=Q:X1=X:Y1=Y:U1=U:V1=V
620 NEXT N:NEXT M
630 A(X,Y)=A(U,V):A(U,V)=C:REM RESTORE
    MOVED PIECE
640 RETURN
650 REM **SCREEN DISPLAY **
```

```
660 FOR I=1 TO 3:FOR J=1 TO 3
670 POSITION 2*I+5,2*(4-J)+4
680 IF A(I,J)=0 THEN PRINT #6;"."
690 IF A(I,J)=+1 THEN PRINT #6;"X"
700 IF A(I,J)=-1 THEN PRINT #6;"O"
710 NEXT J:NEXT I
720 POSITION 0,10
730 RETURN
740 SOUND 0,100,10,6:FOR I=1 TO 100:NEXT
    I:SOUND 0,0,0,0:END
750 TRAP 40000:GOTO 330
```

Type in the program and RUN it. The computer handles all the moves correctly, but note that if you cannot make a move you will have to terminate the game with the BREAK key.

Enter your moves in the form (FROM) X,Y (return) (TO) U,V (return), where X,Y are the starting positions of your pawn – X is the horizontal coordinate and Y the vertical – and U,V are where it moves to. For example, type in (FROM) 1,1 (return) (TO) 1,2 (return).

Lines 110 to 140 clear the Hexapawn board, which is stored in array A(3,3). Lines 160 and 170 set up the pieces which are displayed in the screen subroutine (lines 660-730).

Your move is entered in lines 340 and 350. The next lines down to line 410 check that your input is legal; if not, you try again. Line 420 moves your legal move on the board and line 440 checks to see whether your move wins.

When it is the program's turn to move, it checks every square of the board to see if it has a piece there (lines 240-260). If there is (board array square is +1), it goes to the move generator, lines 470-530.

The square where the program piece was found was at array I,J. These locations are temporarily stored in locations X and Y. Square X,Y represents where the piece is. It then reduces Y by 1, since the piece is moving down the board, and X is varied between −1,0 and +1. The new possible moves are stored in variables U and V.

The program tests to see if it can move forward (U=X), if no other piece obstructs it, and if it can move diagonally forward (U=X+1 or U=X−1), provided that there is an opposing piece to capture.

If any of these conditions are met, the program goes to the evaluation subroutine (lines 550-640). The piece at X,Y is temporarily moved to square U,V, the position is evaluated and scored (score in variable Q), and the piece is move back to its original square, replacing any temporarily captured piece that was at square U,V. (Don't ever forget to do that!)

The evaluation function tests for material count – captures are

favoured – and for the distance that each piece on both sides has advanced (lines 570 and 580). A small random factor is added for variety.

The scores for each move are then matched with the store QQ, which was previously set at −100. If the score exceeds QQ, the moves leading to that score are stored in variables X1,Y1,U1 and V1.

When the move routine is completed, the value of QQ is examined. If it is still −100, then the program made no moves and the result is stalemate (line 270). Otherwise, the piece at square X1,Y1 is moved to square U1,V1 (line 280). If the piece is on the last rank, then the program has won (line 300), otherwise it is your turn to move again.

Try to understand the program Hexapawn before moving on to the next chapter which contains two more program examples.

CHAPTER 7
Game Examples

It is my intention in this chapter to show how a conventional evaluation function, as used in games of chess or draughts, can give perfect play in simpler games with minimal effort. The principle is that the program generates each legal move for itself, then scores the position which arises after each of these legal moves has been made. The highest scoring (best) move is the one played. Two games will be considered.

Both games may look quite long. The problem with all such computer programs is the amount of effort devoted to the screen display and error-trapping for the user's input. The most important part of both programs is the 1-ply evaluation function.

Noughts and Crosses (Tic-Tac-Toe)

Noughts and crosses is an extremely simple game played on a 3*3 board. For the sake of convenience, I shall label the squares of the board as follows:

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

It is not possible for either side to win against best play by the other.

Previous approaches to the game include the 'if there's an enemy piece here, then you put your piece there' method. The method can just be successful owing to the comparatively small number of moves which need to be considered. The total number of all possible moves is 9! (factorial 9) which comes to 362,880 moves. Since the board exhibits four-fold symmetry, it is in principle possible to reduce the number of positions which have to be considered.

Different positions are said to show *symmetry* if they can be transposed into each other by rotating them. For example, after playing an X into square 5, then an O placed in any of the squares 1, 3, 7, or 9 will lead to the same position, as will an O played into squares 2, 4, 6 or 8. However, such means of simplification have not been much employed by the practitioners of the method.

Another ingenious method of solving the noughts and crosses problem is that of making the machine play randomly, remembering the moves which lead to defeat and avoiding these move sequences thereafter. This method is applicable only to very simple games, such as noughts and crosses and hexapawn, owing to the very large number of moves that may have to be stored in some games.

Noughts and crosses can, in principle, be played by a combination of an evaluation function coupled with the ability to 'look-ahead' – evaluate the opponent responses to its moves. It is the latter which takes most of the time as the game 'tree' of possible moves and counter-moves grows geometrically; in this particular case, the geometric growth is restrained by the limited number of remaining legal moves. However, noughts and crosses is so simple that it is possible to play a perfect game with a 1-ply search.

Another attraction of using an evaluation function to assign scores to each possible move is that of drawn games – where no legal moves remain – and wins for either side can be easily recognised with no extra work.

To win at noughts and crosses, you require a straight line – up, down or diagonally – of three noughts or three crosses; there are eight such lines in all. Assigning a score of +1 to each machine piece and −1 to each opponent piece means that a line score of +3 wins for the machine and of −2 wins for the opponent (since he will have the next move). A machine win must be given priority over the projected win for the opponent.

A positive score in any line favours the program, a negative score favours the opponent. The sum of all the line scores, weighted for 2 or 3 pieces of one type in a line, gives the evaluation score which is, in essence, a measure of the control which the machine has over all the lines.

This simple evaluation function is found in lines 560-710, and it plays a perfect game.

The method of play is that the machine scores each of its possible legal moves, and stores the best score (Q) yet found in the store SC together with the move creating it in store K. Counter CO is incremented whenever a legal move can be made; if it remains at 0 then there are no legal moves and the game is ended.

If the program moves first, it will always calculate that it should occupy the centre square (greatest line control), so I have given it this move as a book opening in line 170. Another book line is found in line 430. This uses the principle of symmetry to recognise a position where the opponent has played into each corner and the program has played into the centre. The EF would force the program to play a losing move into a corner.

The random number generator in line 700 causes random selection between moves of otherwise equal merit. Thus, if the opponent starts in square 5, the machine will randomly play between squares 1, 3, 7 and 9.

The program takes less than five seconds to generate each move. It will

never lose. It will also force a win against incorrect play by the opponent even as early as his first move.

Type in the Noughts and Crosses program and RUN it. After deciding whether to move first, simply make your move, on your turn, by entering the number of the square where you wish to place your O and X. The program handles everything else.

```
100 REM ** NOUGHTS AND CROSSES **
110 DIM A(9),Q(9),A$(9),C$(1),D$(1)
120 GRAPHICS 0
130 POSITION 10,6:PRINT "NOUGHTS AND
    CROSSES"
140 FOR I=1 TO 9:A(I)=0:A$(I,I)=STR$(I):
    NEXT I
150 POSITION 7,10:PRINT "DO WANT TO GO
    FIRST(Y/N)?"
160 INPUT C$:IF C$="Y" THEN C$="X":D$="O":
    GOTO 180
170 A(5)=1:A$(5,5)="X":C$="O":D$="X"
180 GRAPHICS 1:POKE 752,1:CO=1:GOTO 300
190 SC=-1000
200 GOSUB 450
210 FOR J=1 TO 9
220 IF A(J)<>0 THEN 250
230 A(J)=1:GOSUB 560:A(J)=0:REM TEMPORARILY
    MOVE PIECE, EVALUATE AND RESTORE PIECE
240 IF Q>SC THEN SC=Q:K=J
250 NEXT J
260 CO=0:A(K)=1:A$(K,K)=D$
270 FOR I=1 TO 9
280 IF A(I)=0 THEN CO=CO+1
290 NEXT I
300 GOSUB 450
310 X=20:GOSUB 770
320 POSITION 0,17
330 IF SC>50 THEN PRINT #6;"
    I WIN!":GOTO 780
340 IF CO=0 THEN PRINT #6;"    END OF GAME":
    GOTO 780
350 POSITION 0,17:? #6;"YOUR MOVE     "
360 GOSUB 720:Z=LETTER-48
370 IF A(Z)<>0 THEN 360
380 A(Z)=-1:A$(Z,Z)=C$
```

```
390 X=10:GOSUB 770
400 POSITION 0,17:? #6;"                "
410 GOSUB 560
420 IF Q<-500 THEN POSITION 0,17:PRINT #6;"
    YOU WIN!":GOTO 780
430 IF INT(Q)=-2 AND (Q(7)=-1 OR Q(8)=-1)
    THEN A(4)=1:A$(4,4)=D$:GOTO 260
440 GOTO 190
450 REM ** SCREEN DISPLAY **
460 M=0:FOR J=5 TO 11 STEP 3
470 FOR I=5 TO 11 STEP 3
480 M=M+1:W=ASC(A$(M,M))
490 IF W=88 THEN COLOR W+32:REM LETTER X
500 IF W=79 THEN COLOR W+160:REM LETTER O
510 IF W<58 AND W>48 THEN COLOR W
520 PLOT I,J
530 NEXT I:NEXT J
540 RETURN
550 REM ** EVALUATE **
560 FOR I=1 TO 3
570 Q(I)=A(I)+A(I+3)+A(I+6):REM VERTICAL
    LINE
580 Q(I+3)=A(3*I)+A(3*I-1)+A(3*I-2):REM
    HORIZONTAL LINE
590 NEXT I
600 Q(7)=A(1)+A(5)+A(9)
610 Q(8)=A(3)+A(5)+A(7):REM DIAGONAL LINES
620 Q=0
630 FOR I=1 TO 8
640 Q=Q+Q(I)
650 IF Q(I)=-3 THEN Q=Q-900
660 IF Q(I)=-2 THEN Q=Q-30
670 IF Q(I)=+2 THEN Q=Q+10
680 IF Q(I)=+3 THEN Q=Q+100
690 NEXT I
700 Q=Q+RND(1)
710 RETURN
720 CLOSE #1:OPEN #1,4,0,"K:"
730 GET #1,LETTER
740 POSITION 10,17:? #6;LETTER-48;"    "
750 IF LETTER<48 OR LETTER>57 THEN 720
760 RETURN
770 SOUND 0,X,12,4:FOR M=1 TO 40:NEXT M:
    SOUND 0,0,0,0:RETURN
```

```
780 SOUND 0,120,10,6:SOUND 1,160,10,6:FOR
    K=1 TO 100:NEXT K:SOUND 0,0,0,0:SOUND
    1,0,0,0:END
```

Race!

Race! is a simulation of the children's game Ludo, retaining the latter's essential features. Just for fun, I have structured the program – it runs so fast that there is no need to worry about the stacking of subroutines.

The rules of Race! are simple. Each side has four pieces, numbered 1 to 4. The computer simulates throwing a die in line 360, giving a random number between 1 and 6. This is displayed. (A die, by the way, is the singular form of dice; one die, two dice).

Both sides need to throw a six to start. After throwing a six, they may place any piece on the 'board', which is actually just a straight line 18 squares long. Each side also has another turn after throwing a six.

If one piece lands on a square occupied by an enemy piece, then the enemy piece returns to the start and has to throw a six to move again.

If two, or more, pieces of the same side reside on one square, then there is a 'block' and no enemy piece can land on, or pass through, the block.

When a piece lands exactly on the end of the board, it is 'home' and is placed in its home location. When all four pieces of one side are home, then that side has won and the game is over.

There are two ways of making the program start moving a piece by throwing a six. One is an elaborate test of the location of each piece when a six is thrown, then moving the piece.

The other method, used in this program, is to put an artificial 'block' on the first five squares of the board which a piece can move over, but not land on. Thus, only a six will clear the artificial block. But now, there are five unused squares on the board. I suppressed these by moving the whole board down five squares. See, for example, line 600. Thus, the board on your TV screen looks as though it is 18 squares long, but it is actually 23 squares long.

Variable Y holds the original location of each piece as its move is tested and V is where it moves to. Variable X holds the original position of the piece which has the best move and therefore actually moves, and U is the square to which it moves. Note that a piece table (A$) is used to hold the position of all the pieces for both sides.

Type in the program and RUN it. Enter your moves by keying in the number of the piece which you want to move, followed by RETURN. If you have no legal move, type in zero (0).

Problem: In what way does this program differ from Ludo? How does it compare with Backgammon?

```
100 REM ** RACE! by John White **
110 INIT=260:THROW=360:CMOVE=410:MOVE=530:
    YMOVE=640:TESTMOVE=780:EVALUATE=860:CL
EAR=960:BEEP=1020
120 HOME=1120:BOOP=1050
130 GOSUB INIT
140 AO=+1:GOSUB THROW
150 GOSUB CMOVE
160 IF QQ=-999 THEN 180
170 GOSUB MOVE
180 IF DIE=6 THEN 140
190 AO=-1:GOSUB THROW
200 GOSUB YMOVE
210 IF J=0 THEN 230
220 GOSUB MOVE
230 IF DIE=6 THEN 190
240 GOTO 140
250 REM ** INITIALISE **
260 DIM A(24),N(3),A$(16),U$(2),V$(8)
270 FOR I=1 TO 24:A(I)=0:NEXT I
280 FOR I=1 TO 5:A(I)=5:NEXT I
290 N(1)=0:N(3)=0:A$="000000000000000":V$=
    "YOUR  MY"
300 GRAPHICS 1:POKE 752,1
310 COLOR 11:PLOT 1,8:DRAWTO 17,8:COLOR 42:
    PLOT 18,8
320 FOR I=1 TO 4:COLOR I+48:PLOT 0,8+I:
    COLOR I+144:PLOT 0,8-I:NEXT I
330 POSITION 7,0:PRINT #6;"RACE!"
340 RETURN
350 REM ** THROW DICE **
360 DIE=INT(6*RND(1)+1)
370 POSITION 2,15:? #6;V$((AO+1)*2+1,(AO+1)
    *2+4);" THROW = ";DIE
380 IF DIE=6 THEN GOSUB BEEP
390 RETURN
400 REM ** COMPUTER MOVE **
410 QQ=-999:FOR K=1 TO 250:NEXT K
420 FOR I=1 TO 4
430 Y=VAL(A$(2*I-1,2*I))
440 IF Y=61 THEN 500:REM PIECE IS HOME
450 V=Y+DIE
460 NG=0:GOSUB TESTMOVE
470 IF NG<>0 THEN 500
```

```
480 GOSUB EVALUATE
490 IF Q>QQ THEN QQ=Q:J=I:X=Y:U=V
500 NEXT I
510 RETURN
520 REM ** MOVE PIECE **
530 U$=STR$(U):IF LEN(U$)=1 THEN U$="0":U$
    (LEN(U$)+1)=STR$(U)
540 IF A(U)=-A0 THEN A(U)=0:GOSUB CLEAR
550 A(U)=A0+A(U)
560 A(X)=A(X)-A0:IF A(X)*A0<0 THEN A(X)=0
570 A$(2*J-1,2*J)=U$
580 IF U=23 THEN GOSUB HOME:RETURN
590 IF X=0 THEN X=5
600 IF A0=1 THEN COLOR J+48:PLOT U-5,8+J:
    COLOR 0:PLOT X-5,8+J
610 IF A0=-1 THEN COLOR J-4+144:PLOT U-5,
    8+I:COLOR 0:PLOT X-5,8+I
620 RETURN
630 REM ** YOUR MOVE **
640 TRAP 1200:PRINT "WHICH PIECE? "
650 INPUT I
660 IF I=0 THEN PRINT "NO MOVES ":J=0:
    RETURN
670 IF I<1 OR I>4 THEN GOSUB BOOP:PRINT
    "SELECT NO. FROM 1 - 4":GOTO 650
680 Y=VAL(A$(2*I+7,2*I+8))
690 IF Y=0 OR Y=61 THEN 710
700 IF A(Y)>=0 THEN GOSUB BOOP:GOTO 650
710 V=Y+DIE
720 NG=0:GOSUB TESTMOVE
730 IF NG<>0 THEN GOSUB BOOP:GOTO 650
740 J=I+4:U=V:X=Y
750 I=I*A0
760 RETURN
770 REM ** TESTMOVE **
780 IF V>23 THEN NG=1:RETURN
790 FOR N=Y+1 TO V
800 IF A(N)=5 THEN 820
810 IF A(N)*A0<-1 THEN NG=1
820 NEXT N
830 IF A(V)=5 THEN NG=1
840 RETURN
850 REM ** EVALUATE **
860 Q=0
```

```
870 IF A(V)=-1 THEN Q=Q+40
880 IF A(V)=+1 THEN Q=Q+20
890 IF A(Y)=+1 THEN Q=Q-10
900 Q=Q+V
910 IF V=6 THEN Q=Q+5
920 IF V=23 THEN Q=Q+40
930 Q=Q+RND(1)
940 RETURN
950 REM ** CLEAR CAPTURED PIECE **
960 M1=0:IF AO=+1 THEN M1=4
970 FOR M=1+M1 TO 4+M1
980 IF A$(2*M-1,2*M)=U$ THEN A$(2*M-1,2*M)=
    "00":GOSUB 1080
990 NEXT M
1000 RETURN
1010 REM BEEP
1020 SOUND 0,80,10,6:FOR K=1 TO 40:NEXT K:
     SOUND 0,0,0,0
1030 RETURN
1040 REM ** BOOP **
1050 SOUND 0,120,10,6:FOR K=1 TO 80:NEXT K:
     SOUND 0,0,0,0
1060 RETURN
1070 REM ** REPLACE CAPTURED PIECE **
1080 IF AO=1 THEN COLOR M-4+144:PLOT 0,
     8-M+4:COLOR 0:PLOT U-5,8-M+4
1090 IF AO=-1 THEN COLOR M+48:PLOT 0,8+M:
     COLOR 0:PLOT U-5,8+M
1100 RETURN
1110 REM ** PIECE HOME **
1120 POSITION 2,15:PRINT #6;"PIECE HOME
     ":A(23)=0:A$(2*J-1,2*J)="61":N(AO+2)=
     N(AO+2)+1
1130 GOSUB BOOP
1140 IF AO=1 THEN COLOR J+48:PLOT 19,12+J:
     COLOR 0:PLOT X-5,8+J
1150 IF AO=-1 THEN COLOR J-4+144:PLOT 19,4+I
     :COLOR 0:PLOT X-5,8+I
1160 IF N(AO+2)<4 THEN RETURN
1170 POSITION 4,13:? #6;" GAME OVER"
1180 GOSUB BOOP
1190 END
1200 TRAP 40000:GOSUB BOOP:GOTO 640
```

CHAPTER 8
Book Openings and the Opening

So far, we have considered methods to enable a program to make those moves which are in general found in the middle game.

A special situation arises in the early moves of a strategy game. Since the starting position for most such games is always known, unlike positions which arise in midgame, it is possible for the programmer to incorporate certain book moves into the opening play. These are moves which are made, not by calculation, but by reference to a 'book opening library' which is loaded with the rest of the program into RAM.

There are three advantages to a book opening library. First, it enables the program to make its moves almost at once, saving time for computation of later moves. Secondly, it enables the program to make certain non-obvious strategic moves, which long experience between human players will have determined to be the best. Thirdly, the play from the library may enable the program to avoid opening traps.

Book openings can be useful in a variety of games, but by far the most important example is chess, where literally hundreds of openings are known. Two well-known examples, from the Queen's Gambit opening, provide illustration. After the Queen's Gambit moves

1. d2–d4 d7–d5
2. c2–c4

then acceptance of the proferred pawn by the program, playing

2.....d5xc4

followed by a grim determination to hang on to it, invariably spells long-term disaster against accurate play by white, the opener. However, it takes a deep search to see this, deeper than any normal chess program can see. As a result, any chess program which is not pre-programmed to avoid the trap invariably falls into it.

The second example from the Queen's Pawn opening is the thematic pawn move c7–c5 by black at some stage. The theoretical justification is to give black space on the queen's side, or sometimes to smash open the

centre for a fianchettoed bishop on black's KN2 (g7) square.

In the Tarrasch line, this thematic move is played as early as move 3:

1. d2–d4 d7–d5
2. c2–c4 e7–e6
3. b1–c3 c7–c5

Yet it is very hard to design a chess program to play this move early in the game by calculation. Directly telling the program to play the move, as part of its book library, is much easier.

The book opening library is so useful that now few games of strategy are played without it. Some commercial libraries are enormous; the Gruenfeld chess cartridge devotes 12K of ROM to openings alone, while some mainframe programs require large disks to hold the entire library. Even programs available in software are now often enhanced by the addition of several book moves, although amateur programmers seem to have trouble with this and rarely program a book line more than three moves deep.

Let us consider what is required of a good book opening library – requirements which many commercial programs do not yet meet.

Firstly, the program should be capable of playing from both ends of the game board, regardless of who moves first. Few game positions are symmetrical in the sense that the same book opening library can be used to play both ends of the chess board. Instead, the program must have the capability to transpose moves from one side of the board to the other. For example, if a book library requires a knight to be moved from square b8 to c6, then the library must also store a routine so that the knight can be made to make the same move from square g1 to f3, should the opponent be playing the same book line from his side of the board.

The general transposition routine is

$$A(x,y) = A(9-x,9-y)$$

where x and y are the positions of the piece on an 8*8 board. It is not necessary to transpose the entire book library, only the moves as they are accepted from the opponent and as they are made by the book. It will also be necessary to move the move-counter up or down by one, depending on which side starts.

The second feature of a book opening library – which is where most amateur libraries fall down – is that the program should be able to distinguish several possible opponent responses which lead to different variations of the same opening, and then switch between them. If the program opens with e2–e4, then opponent responses of e7–e5 (Ruy Lopez) and c7–c5 (Sicilian) are highly likely – to say the least – as well as numerous other possibilities. If the program has decided that it will only play the Sicilian on this occasion, then the reply e7–e5 will

immediately take the program out of the book library even if the move was part of the original book. At a more subtle level, the Sicilian defence has several variants – for example the Rauzer and Dragon lines – and these must also be distinguished. I know of several commercial programs which cannot switch between openings in this way.

A third feature of a good book opening is that the user should be able to select his own opening at will; failing a choice, the book should pick an opening at random. The program should be able to prompt the opponent as to his continuation in the selected book line.

Finally, when the library is exhausted or the user deviates from the library line and its stored alternatives, the program must return to its own evaluation function.

Construction of a book library is by no means a trivial task. The size of the library is essentially dependent only on the size of available free memory.

The moves made by the library should certainly lead the program into a position which it likes. Many chess programs show a strong preference for pins, as of knights against kings by bishops. In this case, suitable book openings include, for example, the Nimzo-Indian line as black:

1. d2–d4 g8–f6
2. c2–c4 e7–e6
3. b1–c3 f8–b4

or the Ruy Lopez as white:

1. e2–e4 e7–e5
2. g1–f3 b8–c6
3. f1–b5

One writer of a famous program, a weak chess player himself, arranged a book library of sorts by asking for a selection of moves from a weak club player. What he got was a book which gave no consideration to its idiosyncratic style of play, and included such curiosities as (program black):

1. d2–d d7–d5
2. c2–c4 g8–f6 ?!

The book library must also leave the program in a sensible position where it knows what to do if a non-book move is made. A classic example of neglecting this precaution came with the old Challenger 7 computers which sometimes tried to play the Cambridge Springs defence to the Queen's Pawn Opening. The line is (computer plays black):

1. d2–d4 d7–d5
2. c2–c4 e7–e6
3. b1–c3 g8–f6
4. c1–g5 b8–d7
5. e2–e3 c7–c6
6. g1–f3 d8–a5

The crux of the position is after black's move 4, where white cannot win a pawn by

5. c4xd5 e6xd5
6. c3xd5

owing to the devastating and striking queen sacrifice:

6.....f6xd5
7. g5xd8 f8–b4+
8. d1–d2 b4xd2
9. e1xd2 e8xd8

when black is winning. And yet, the Challenger 7 could not 'see' all of moves 5 to 9. After white plays

5. c4xd5

it will not sacrifice its queen in the approved manner. Thus, the manufacturer's choice of this opening in effect presented the human opponent, white, with a pawn for free!

Even modern machines are not free from this problem. The latest Steinitz chess module makes a similar present of a pawn to white in one line of the Benoni, a line where the pawn capture should be unsound, costing white a piece.

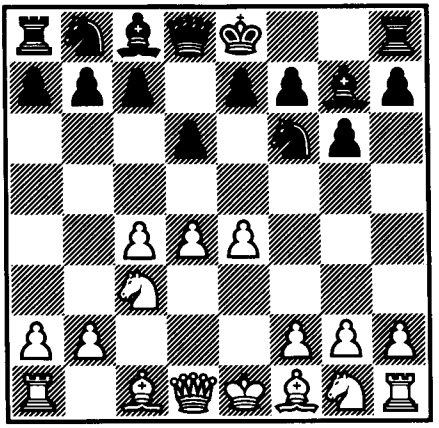
The manufacturers of the Chess Champion Mark V computer have elected to pre-program a wide range of highly unusual book openings which soon flummox both human and machine opponents. However, most good commercial book libraries give random conventional responses, often weighted so as to mimic the frequency that different openings are encountered in actual human play. The latter approach is, in my opinion, preferable. The shortcomings of unusual book lines can be mercilessly exposed by a real expert. In any case, most players would rather practise against openings that they will also encounter away from the computer.

Another oversight by many chess programmers is what to do against a very odd opening, such as a2–a4. Most libraries immediately give up and

the program enters its evaluation mode. In fact, a library can automatically play e7–e5 against any opening move except d2–d4, f2–f4 or g1–f3. The general concept of having a 'universal' opening move, allowing for certain exceptions, is widely applicable to most games of strategy.

The last problem to consider is whether to store a book opening library in the form of moves or positions. The advantage of the former is that it is faster and requires less space. The advantage of the latter is that it is possible to reach book positions by transportation of move order.

Diagram 8-1



For example, we can reach the standard King's-Indian positions in **Diagram 8-1** by several routes:

eg 1. d2–d4 g8–f6
2. c2–c4 g7–g6
3. b1–c3 f8–g7
4. e2–e4 d7–d6

or 1. d2–d4 d7–d6
2. c2–c4 g8–f6
3. b1–c3 g7–g6
4. e2–e4 f8–g7

or 1. c2–c4 g8–f6
2. b1–c3 d7–d6
3. d2–d4 g7–g6
4. e2–e4 f8–g7 and so on.

If the program is to make a book move from position 8–1, then either the library must store all the likely moves to that position or it can store the position itself and the move from it.

Both appearances are viable at machine code speeds, but calculation of positions is too slow in BASIC. The approaches are amenable to different programming methods.

A book library is *not* constructed by storing a series of moves as a geometrically increasing tree. The method of 'He made move A, so I made move B, so he made C, so I shall make move D' is tremendously wasteful of memory space, and, at deep levels into the tree, the time of search becomes excessive even at machine code speeds.

What is required is a move-matching method which takes the opponent move, finds the move number and at once comes up with the book move. The important feature is that no matter how deep into the book that the program looks, the time to find the response remains short. (In fact the time scarcely varies regardless of the depth).

The method used by all good libraries is essentially based on the principle known as 'hash coding'. A full account of hash coding is beyond the scope of this book, but the general idea is that the opponent move or the position arising is assigned a distinctive hash code which is calculated on one of its properties. The program then takes the code and looks up what happens next from a large table. As an example of hashing, let us take a series of people's names: JOHN, JOE, FRED, MARY and TOM. One way of assigning unique hash codes to each name would be to sum the ASCII values of the letters of each name. Thus JOHN becomes ASC("J")+ASC("O")+ASC("H")+ASC("N") = 303

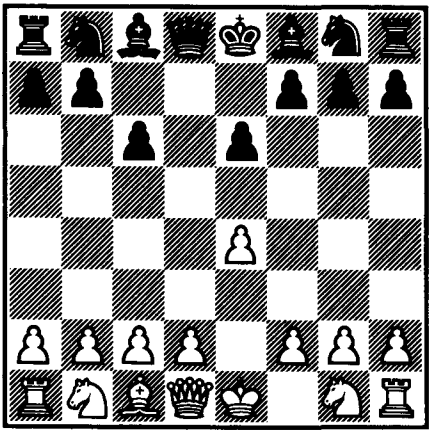
We could then find the ages of JOHN, JOE, FRED, etc. by looking up in a table what age value is assigned to hash code 303 (for JOHN).

But what if there are two people, called JOHN and NHOJ? They will have the same hash code values according to the formula above, resulting in a 'collision'. Most of the theory of hash coding is concerned with what to do in the event of collisions, and a full account can be found in the magazine *Practical Computing* (September 1982) for BASIC programmers, and in the book *6502 Assembly Language* by Rodney Zaks (Sybex) for assembly language programmers.

Hash coding is particularly suited to book opening libraries based on positions, rather than moves. It is easier to implement if all the pieces are already stored in a piece table, when the positions of the pieces in the table also indicate the nature of the piece. It also has a wider applicability.

The position in **Diagram 8-2** was given in the *Daily Telegraph* newspaper as a puzzle. Position 8-2 must be reached in *exactly* 4 moves by white and 4 moves by black from the normal chess starting position. (Try it. There are no catches, only legal moves are required, but it is extremely difficult).

Diagram 8-2



The solution will be given for the frustrated at the end of the chapter. It reputedly took grandmaster Petrosian 20 minutes.

Hash coding is the key to using a computer to find the moves. No evaluation function is needed, nor can any pruning be permitted since all legal moves must be tested for both sides. The program should generate each legal move for both sides alternately until both sides have made all possible permutations of exactly four moves. Each position that arises is hashed and compared with the wanted hash value previously determined for the position in Diagram 8-2. When a match is found, the move sequence leading to it gives the solution to the puzzle.

The moves of a book opening library can be semi-hashed – if that is the right expression – so as to give a match of moves versus move number. These could be stored in a two-dimensional string array. Since neither Atari BASIC nor machine code accommodate two-dimensional string arrays, we are forced back to the one-dimensional vectors described in Chapter 5.

This approach is illustrated in the Book Opening program given at the end of the chapter. The string array B$ stores all the opening moves for the computer. Array C$ stores all the corresponding moves for the opponent. The DATA statements in lines 270-460 store all the openings in numerical code. Each DATA statement is made up of blocks of four figures, such as 7866. Each block refers to a board location: from square 7,8 to square 6,6 (knight move). Note the blocks of zeros before and closing the book.

The principle of the program is that the computer first chooses its opening line OP, either randomly or by selection by the opponent. The program makes its move, then gets the opponent response. If the response is correct for the line OP, then the next book move is made.

If the response does not fit, it is matched with other book lines which also made the same last program move and accepted the same opponent response. The book line indicator OP is then updated to the new line, if a match is found. Otherwise, the book library ends.

One difficulty with this simplified approach to storing book openings is that it is sometimes possible to 'fool' the program by arriving at the same consecutive three moves, on the same turns, from different positions.

The difficulty can be obviated to some extent by careful design of the library. Another partial solution is to check that there is actually a piece at the square from which the book library is trying to make its next move. In the Book Opening program, the check can be seen in line 1030.

A more sophisticated, but still partial, solution is to store a token representing the value of each piece with each book opening move. It then becomes possible to check that the appropriate piece is present to be moved by the book library.

More complex solutions to the difficulty are not worth pursuing, since hash coding is now more suitable.

I have often used the method illustrated in the program to provide an opening library for my strategy games, and have found it generally satisfactory.

The program is purely illustrative, and ten book lines are included which can be selected by number (see **Table 8-1**). Alternatively, select book opening 'O' and the program will select an opening randomly.

Enter your move in the form C7C5(Return). If you want a hint, type in HELP(Return).

The chess pieces used in the program are those that were described in Table 5-1.

Table 8-1

Ref No	Opening
0	Random Selection by Program
1	Sicilian – Rauzer
2	Sicilian – Dragon
3	Ruy Lopez – Open
4	Giuoco Piano
5	King's Gambit
6	French – Winawer
7	QGD – Orthodox
8	Nimzo-Indian – Rubinstein
9	King's Indian – Classical
10	English – Symmetrical

The main book move checking routines are found in lines 920-1040. Castling is stored as a king move only. Extra program routines recognise the odd king move, and complete the castling.

In this illustrative program, the book DATA statements occupy only a small fraction of the total program lines. In a full book, they would dwarf the rest of the program. If you understand the program, try modifying it to accept the book moves as DATA from disc or tape. You could then build up a library of book openings entitled Nimzo-Indian, Ruy Lopez and so on, which would give you a graphical means of learning book openings – much more fun than learning from books. In this connection, note that the book moves are stored in DATA statements that are Z blocks long – where Z is the total number of lines stored – and each block has four numbers, and that there are two DATA statements for each move of the line (one for each side).

The book terminates as soon as a block of '0000' is encountered in the line. Thus some lines could be shortened relative to the others, just by filling the unwanted part of the line with zeros.

Opening development

It will often happen that the program exits from its book library while it is still in the opening. While this may be of no consequence in many cases, in some games, such as chess, it is important that the program use a *modified* evaluation function until the opening moves are over. The opening stage will probably be complete after the first ten moves or so, necessitating counting the moves as the program makes them.

In the case of chess, the modified EF should give weight to developing the minor pieces (knights before bishops), castling and to control of the central squares with pawns. According to Levy, development of the queen before castling should be subject to a penalty of half a pawn; if you adopt this, make sure that your penalty for doubling pawns on a file is more than half a pawn.

Always consider what will happen in the opening of your program if the opponent steps quickly out of the book library – perhaps deliberately.

```
100 REM ** BOOK OPENING **
110 GRAPHICS 0:PRINT "       BOOK OPENING"
120 Z=10:REM NO. OF OPENINGS STORED
130 M=8:REM DEPTH OF OPENINGS STORED
140 PRINT "Storing positions."
150 DIM A(8,8),X(4),Y(4)
160 DIM B$((Z+2)*(M+1)*4),C$((Z+2)*(M+1)*4),
    X$(4),Y$(4),H$(4),OP$(Z)
170 DIM A$(Z*4),G$(37),I$(42):C$="":B$=""
180 GOSUB 1060
```

```
190 RESTORE 220
200 FOR I=1 TO 8:FOR J=1 TO 8:A(I,J)=0:NEXT
    J:NEXT I
210 FOR I=1 TO 8:READ C:A(I,1)=C:A(I,8)=A(I,
    1)+32:A(I,7)=34:A(I,2)=2:NEXT I
220 DATA 5,3,4,6,7,4,3,5
230 FOR I=1 TO M+2
240 READ A$:B$(LEN(B$)+1)=A$
250 READ A$:C$(LEN(C$)+1)=A$
260 NEXT I
270 DATA 000000000000000000000000000000000
    000000
280 DATA 000000000000000000000000000000000
    0000000000000000
290 DATA 525452545254525452545254244244244
    4432 34
300 DATA 373537355755575557555755647457866
    78 663735
310 DATA 71637163716371636264424432343234323
    42133
320 DATA 474647462836283655644745575657567
    762836
330 DATA 424442446125613471632133213321332
    37273
340 DATA 354435441716683577756824786668246
    777776
350 DATA 634463442514323361345455317552535
    2 546172
360 DATA 786678667866786647463735685727264
    7 466877
370 DATA 213321335171424451711213716361436
    1 527163
380 DATA 283677766857554487862433587838275
    8 787866
390 DATA 317531536151334442442233525371637
    1 635171
400 DATA 384768772725352468777857284724335
    7 555878
410 DATA 414261521423314232334174113122335
    1 714244
420 DATA 183828364746244228363544373647452
    8 473544
430 DATA 513151713233214272737477614331134
    4 456344
```

```
440 DATA 26445878587847457574887845342847473
    53644
450 DATA 000000000000000000000000000000000000
    000000
460 DATA 000000000000000000000000000000000000
    000000
470 H$="0000":M0=0:B0=1
480 POSITION 5,10:PRINT "SELECT OPENING"
490 INPUT OP:IF OP<0 OR OP>Z THEN 480
500 IF OP=0 THEN OP=INT(RND(1)*10)+1
510 REM ** INITIALISATION COMPLETE **
520 GRAPHICS 2:POKE 752,1:POKE 756,F/256
530 M0=M0+1
540 IF B0<>0 THEN GOSUB 920:IF B0<>0 THEN
    560
550 POSITION 0,20:PRINT "END OF BOOK":SOUND
    0,60,10,6:FOR I=1 TO 60:NEXT I:SOUND
    0,0,0,0:END
560 POSITION 0,0:PRINT #6;"  FROM ";CHR$(X
    (1)+64);",";X(2);" TO ";CHR$(X(3)+64);
    ",";X(4)
570 A(X(3),X(4))=A(X(1),X(2)):A(X(1),X(2))=
    0:IF A(X(3),X(4))=7 THEN R=1:R1=X(3):
    GOSUB 880
580 REM ** SCREEN DISPLAY **
590 FOR J=1 TO 8:FOR I=1 TO 8
600 IF A(9-I,J)=0 THEN COLOR ASC(",")
610 IF A(9-I,J)>0 THEN COLOR A(9-I,J)+128
620 IF A(9-I,J)>10 THEN COLOR A(9-I,J)-32
630 PLOT 2*I,J
640 NEXT I:PRINT :NEXT J
650 POSITION 0,9:PRINT #6;"  H G F E D
    C B A"
660 FOR I=1 TO 8:POSITION 0,I:PRINT #6;I:
    NEXT I
670 REM ** OPPONENT MOVE  **
680 POKE 752,0:TRAP 1050
690 SOUND 0,100,10,6:FOR I=1 TO 40:NEXT I:
    SOUND 0,0,0,0
700 PRINT " YOUR MOVE":INPUT H$
710 IF H$<>"HELP" THEN 750
720 PRINT "BOOK LIBRARY SUGGESTS FROM ";
730 PRINT CHR$(Y(1)+64);",";Y(2);" TO ";
    CHR$(Y(3)+64);",";Y(4)
```

```
740 GOTO 680
750 POKE 752,1
760 A1=VAL(CHR$(ASC(H$(1,1))-16)):A2=VAL
    (CHR$(ASC(H$(3,3))-16))
770 IF A1>8 OR A2>8 THEN 680
780 B1=VAL(H$(2,2)):B2=VAL(H$(4,4))
790 IF B1>8 OR B2>8 THEN 680
800 IF A(A1,B1)<34 OR A(A1,B1)>39 THEN 680
810 A(A2,B2)=A(A1,B1):A(A1,B1)=0
820 H$=STR$(1000*A1+100*B1+10*A2+B2)
830 POSITION (9-A1)*2,B1:PRINT #6;","
840 POSITION (9-A2)*2,B2:PRINT #6;CHR$(A(A2,
    B2)-32)
850 IF A(A2,B2)=39 THEN R=8:R1=A2:GOSUB 880
860 GOTO 530
870 REM ** CASTLING ROUTINE **
880 IF R1=3 THEN A(4,R)=A(1,R):A(1,R)=0
890 IF R1=7 THEN A(6,R)=A(8,R):A(8,R)=0
900 RETURN
910 REM ** BOOK ROUTINE **
920 K=(MO-1)*Z*4+OP*4-3
930 IF H$=C$(K,K+3) THEN 990
940 OP$="":FOR I=1 TO Z:K=(MO-1)*Z*4+I*4-3
950 IF H$=C$(K,K+3) AND B$(K,K+3)=X$ THEN
    OP$(LEN(OP$)+1)=STR$(I)
960 NEXT I
970 IF OP$="" THEN BO=0:RETURN
980 J=INT(RND(1)*LEN(OP$))+1:OP=VAL(OP$
    (J,J))
990 X$=B$(MO*Z*4+OP*4-3,MO*Z*4+OP*4)
1000 Y$=C$(MO*Z*4+OP*4-3,MO*Z*4+OP*4)
1010 FOR I=1 TO 4:X(I)=VAL(X$(I,I)):Y(I)=
     VAL(Y$(I,I)):NEXT I
1020 IF X(1)=0 OR Y(1)=0 THEN BO=0
1030 IF A(X(1),X(2))<1 OR A(X(1),X(2))>10
     THEN BO=0
1040 RETURN
1050 TRAP 32767:GOTO 680
1060 REM ** SET UP CHESS PIECES **
1070 F=(PEEK(106)-8)*256:I$="SQUAREPAWN
     KNIGHTBISHOPROOK  QUEEN KING  "
1080 RESTORE 1110
1090 FOR I=1 TO 36:READ A:POKE ADR(G$)+I,A:
```

```
      NEXT I
1100 REM ** TRANSFER CHARACTER SET TO RAM **
1110 DATA 104,104,133,205,104,133,204,104,
      133,207,104,133,206,104,104,133,208,16
      6,208,160,0
1120 DATA 177,204,145,206,200,208,249,230,
      205,230,207,202,208,240,96
1130 D=USR(ADR(G$)+1,224*256,F,4)
1140 REM ** READ CHESS CHARACTER DATA **
1150 FOR I=0 TO 63:READ A:POKE F+I,A:NEXT I
1160 DATA 0,0,0,0,0,0,0,0
1170 DATA 255,129,129,129,129,129,129,255
1180 DATA 0,0,16,56,56,16,124,0
1190 REM ** PAWN
1200 DATA 0,16,56,120,24,56,124,0
1210 REM ** KNIGHT
1220 DATA 0,16,40,68,108,56,124,0
1230 REM ** BISHOP
1240 DATA 0,84,124,56,56,124,124,0
1250 REM ** ROOK
1260 DATA 0,84,40,16,108,124,124,0
1270 REM ** QUEEN
1280 DATA 0,16,56,16,56,124,124,0
1290 REM ** KING
1300 GRAPHICS 2:POKE 752,1
1310 POKE 756,F/256:REM ** ENABLE NEW
      CHARACTER SET
1320 FOR I=2 TO 7
1330 COLOR I:PLOT 5,I:POSITION 8,1:PRINT #6;
      I$(6*I-5,6*I):COLOR I+128:PLOT 17,I
1340 NEXT I
1350 RETURN
```

The solution to the problem in **Diagram 8-2:**

1. e2–e4 e7–e5
2. f1–b5 e8–e7
3. b5xd7 c7–c6
4. d7–e8 e7xe8

Easy when you know how, isn't it?

CHAPTER 9
Endgame

There comes a stage in most games of strategy when there are few pieces or moves left to play. This stage is generally referred to as the Endgame and usually different programming principles are needed to play it well. We shall again use chess as our example.

By convention, a chess end game is deemed to start when the queens are off the board. However, it is possible to reach a position where practically all the material except the queens is off the board, and this clearly gives an endgame. Conversely, positions also arise when the queens are exchanged very early, leaving most of the other material still on the board. From a programming viewpoint, this is not an endgame.

On of the earliest commercial chess programs to tackle the problem was the famous Sargon 2.5 chess computer, which caused the program to enter its endgame at move 30, regardless of the position. Move 30 was taken as a crude yardstick that the endgame was about to begin.

Subsequent programs, including those from the same manufacturer, have used the EF to decide when the endgame begins, a much more satisfactory arrangement. The total amount of material on the board is evaluated, excluding kings, and the result is tested to see if the pre-set endgame level has been reached. In the Morphy chess program, the endgame is reached when the total material count is less than a value of about 25 (pawn = 1, rook = 5, etc), while Morphy's successor, the Steinitz program, starts its endgame when about 30-35 points worth of material remain.

In BASIC, we would write the subroutine:

```
1010 (DIM A(8,8))
1020 Q = 0: FOR I = 1: FOR J = 1 TO 8
1030 Q = Q + ABS(A(I,J))
1040 NEXT J: NEXT I
1050 Q = Q-2*255: REM REMOVE KINGS
1060 IF Q < 25 THEN (start endgame)
1070 RETURN
```

The reason that endgame routines need to be employed is that different

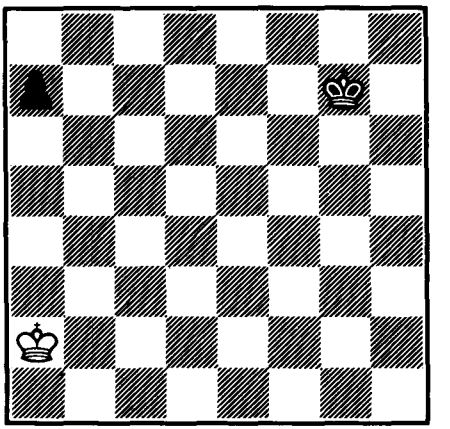
factors now assume prominence which were less important previously. For example, the promotion of a pawn to a queen becomes of critical importance in a chess endgame, but would be unlikely to arise during the middle of the game.

Many programs reduce the time spent on thinking in the closing stages of a game, because there are fewer moves to consider. Instead, it is much wiser for the program to increase its depth of search. This is particularly facile when using the method of search known as iterative deepening, when it is possible to keep increasing the depth of search until either a timer halts the search or a definite, pre-selected number of positions have been evaluated.

A chess program has two disadvantages in the endgame compared with its human adversary. Firstly, because of the limited number of moves, a human can spot the critical line very quickly and mentally search that one line to great depth.

A simple example of this can be seen in **Diagram 9-1.**

Diagram 9-1



It is at once obvious to the human eye that black (to move) *must* play g7–f6 (or to f7 or to f8) to save the pawn. Anything else loses the pawn to white's attacking king. Yet to see this 'obvious' move takes a search of 11 ply for the program. Many chess programs fail the test, either advancing the pawn or aimlessly moving the king. Some, however, move the king in the right direction simply because of a desire to centralise it.

As a matter of detail, black cannot win this endgame even if he does save the pawn. Against accurate play by white, stalemate always occurs.

The second disadvantage is that a human player can immediately recognise chess patterns and play with full knowledge of the moves required by that pattern. It is very difficult to teach a program that

positions 9-2 and 9-3 are essentially the same and can be played the same way.

Diagram 9-2

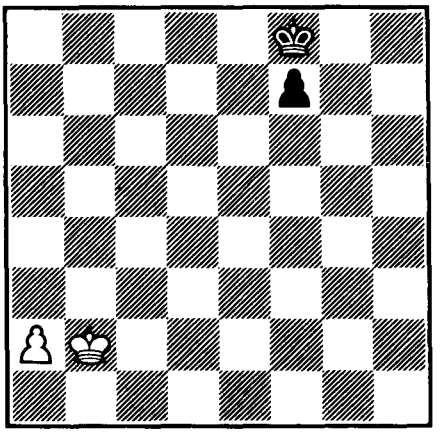
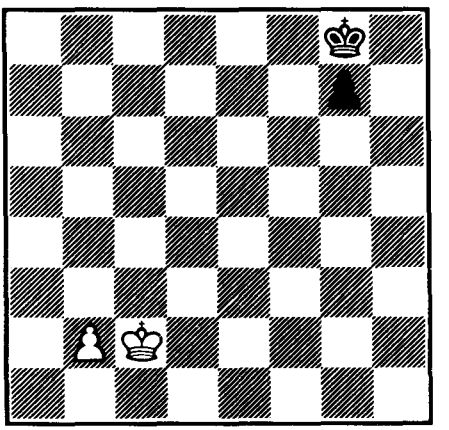


Diagram 9-3



Modifications to the chess evaluation function during the endgame should include the following features:

Firstly, the look-ahead must be increased, as explained above.

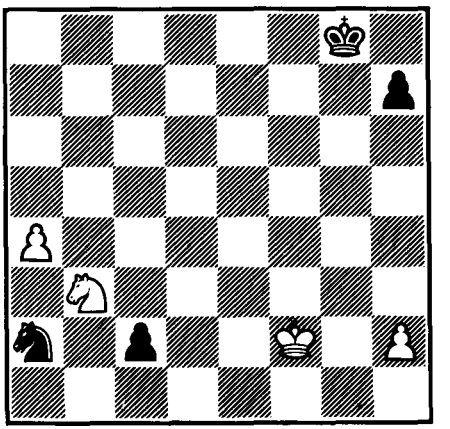
Secondly, the king, which was previously tucked away out of harm's way in one corner (e.g., by castling) must now be made more active. The king is rather a strong piece in the endgame, and has roughly the power of a rook. It should be made to make its way cautiously towards the centre of the board.

Thirdly, pawns should be advanced as chains up the board. There is a great deal of theoretical work published on pawn movements, but this need not concern us here. The value of a pawn should increase as it arrives near its queening square, especially if it is supported by other pawns. This will compensate for the fact that promotion of a strong, advanced pawn to a queen may remain outside the program's normal look-ahead.

The threat or promise of pawn promotion is very difficult to program properly. Levy has recommended that a pawn which cannot be caught by an opposing piece before promotion should be assigned the value of a queen, but it is time-consuming to implement the routines required to see if the pawn can be blocked. Very few chess programs really understand the potential of an advanced pawn until the promotion of the pawn comes within their normal look-ahead of 3 – 4 moves.

The following position arose on move 49 in a game between the Chess Champion Mark V (playing black) and a human (white) in the Silica Chess Computer Symposium of Spring 1982, after the queens had been exchanged on move 7:

Diagram 9-4



White had clear chances to promote his outside 'a' pawn while black fiddled around at the bottom of the board. Unfortunately, the human player did not appreciate the strength of his passed pawn either, and the game was eventually drawn.

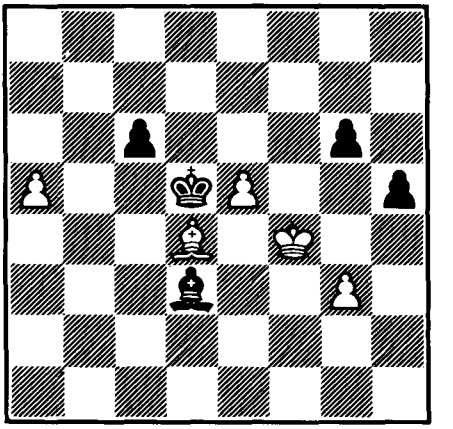
From the diagram, white should have played a4–a5!, which probably forces the computer to play a2–b4. In fact, the program might well have missed that move and played on as it did, as do other chess programs against which I have tested the same position. Instead white played 1. f2–e3 and play continued

1..... c2–c1 (=Q)
2. b3xc1 a2xc1
3. e3–d4 g8–g7

and ultimately the draw.

Another problem on the same lines was described by WW Foster in the magazine *Personal Computing* (October, 1980). In **Diagram 9-5**, with the program (black) to move, the black king must not capture the white bishop; otherwise, one of the white pawns on the 'a' and 'e' files can be pushed home against any black defence. If black refrains from taking the bishop, the position is probably a draw.

Diagram 9-5



To see this result requires a look-ahead of at least 12 ply, and no commercial program can see within a reasonable period of time (several hours!) that it must leave the white bishop alone. Only a special routine would give the answer within a short period of time and it would be hard to program so as to be widely applicable.

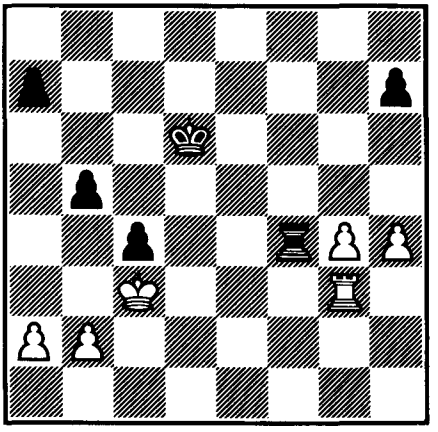
Another of my favourite endgame positions, arising from an actual game, is shown in **Diagram 9-6.**

There is no 'right' answer to the problem of the program's (black's) best move – some strong programs play a7–a6, others move the king downwards – but one move is definitely wrong, despite being favoured by two powerful commercial programs. That move is 1.....h7–h5 ?! which is well met by

2. g4–g5 f4xh4 ??
3. g5–g6!

and black cannot prevent the g pawn from queening except at the loss

Diagram 9-6



of the rook. Once again, a program failed to assess the threat of pawn promotion properly.

Horizon moves also come into prominence in the endgame. Once a program determines that an enemy pawn is about to become a queen, it will often make any sacrifice – giving up several pieces in succession – to delay the fateful move. The reason is that at any one moment the loss of a minor piece (–3 or –5) to the program outweighs the advantage to the opponent of replacing a pawn with a queen (–8 for the program).

A final possibility for playing the endgame is to construct tables of moves in response to certain positions which arise. This requires knowledge of hashing positions, described in the chapter on book openings, and must also allow for transpositions between related positions. Some mainframe computer chess programs have even started from desirable final positions, working back to find a forced move sequence which leads from the present position to the desired one.

The last approach probably requires a large disk to store all the 'book endings' and is hardly suited to the Atari computers. I have never implemented the method myself, but some readers may find the knowledge useful.

The principles which I have expounded for playing the endgame of chess apply equally to numerous other strategy games. At draughts, for example, one of my own commercial programs plays endgame routines when just five pieces remain on the board, at which stage the inferior side is expected to head for a 'double-corner' and the superior side to pin the inferior against the edge of the board.

Always consider the possibility that the endgame play of your program may have different requirements to its play in midgame.

Look-up tables

Reference has been made occasionally to look-up tables in this book and it may help some readers if I give an example of what a look-up table actually does.

It often happens that we want to assign a value to a second variable which is dependent on the value calculated by the program for the first variable. For example, if the program calculates that variable C is equal to 1, we may wish to set variable D equal to 10. If C is equal to 2, we shall make D 20. If C is equal to 3, we put D equal to 30, and so on.

In this trivial case, we can write

```
D = 10 * C
```

But what if there is no direct relationship between C and D? Suppose that when the program calculates C as 1, we wish to put D equal to 100. When C is 2, D is to become 35. When C is 3, D will be 0.1. We can write out the relationship between C and D in the form of a table. **Table 9-1** is called a Look-Up Table. The program looks-up the value of D every time that it calculates the value of C.

Table 9-1

Value of C	Associated Value of D
1	100
2	35
3	0.1
4	61
5	79
6	82
7	3
8	6
9	0
10	1000
.....	.....
.....	.....

Program 9-1 shows how to implement a look-up table in BASIC. Lines 10 to 50 set up the table in array A(X). I have assumed that there are ten elements to the table; that is, ten numbers to look up.

In line 70, the program calculates a value for C. It is a simple random number generator returning values between 1 and 10.

In line 80, the program looks up the value of D in the array A(X). The results for C and D are printed in line 90.

The loop in lines 60 and 100 simply causes the program to look up random values of C twenty times in all.

```
5 REM PROGRAM 9-1
10 DIM A(10)
20 FOR I=1 TO 10
30 READ B:A(I)=B
40 NEXT I:PRINT "C","D"
50 DATA 100,35,0.1,61,79,82,3,6,0,1000
60 FOR I=1 TO 20
70 C=INT(RND(0)*10)+1
80 D=A(C)
90 PRINT C,D
100 NEXT I
```

CHAPTER 10
Computer Draughts

Draughts is an ancient game, known in the USA as Checkers and on the Continent as Dames or Damenspiel. It is played on a normal chess board but is considerably less complex than chess.

The object of draughts is to prevent your opponent from moving, usually achieved by removing all the opponent pieces by jumping over them. There are only two types of piece – men and kings – these have very simple moves. Draughts is therefore eminently suitable for programming onto a computer, although the ability of a piece to make multiple jumps requires a little thought in the move generator. A little known rule is that if a man jumps onto the eighth rank, becoming a king, it is not then able to make further jumps as a king on that move.

It is probably true to say that most competent draughts players are also chess players. There are comparatively few draughts clubs. Many chess players denounce the lack of complexity in draughts, yet there can be no doubt that a great deal of skill is required to play the game well. According to the games expert Hoyle, one former world champion could set up a position and demonstrate a forced win from it in a maximum of 53 (sic) moves.

It seems fairly clear that tactical ability is more important than strategic skill at draughts, the opposite to the requirements of chess. The winning player will be the one who can calculate most deeply into a position. This suits the modern computer which can analyse much deeper into a game of draughts – owing to the limited number of moves – than into chess. The best draughts programs have now won some games against the world human champions.

As is usual when writing computerised games of strategy, it is necessary to consider the evaluation function – EF – to assess a position and to decide how deep the program should look into a position, remembering that the number of moves increases geometrically the deeper the program looks.

The first strong draughts program was written in the USA by Arthur Samuel for the IBM 704 in the early 1950s. It looked to a depth of 3 ply – further for captures – and used a sophisticated EF based on the experience of draughts experts. It was also programmed to exchange

pieces when it was ahead, and to avoid exchanges when it was losing. The alpha-beta pruning mechanism was not known at the time, but the machine was able to play reasonably quickly and already represented a strong opponent to humans.

Samuel proposed a new method of numbering the draughts board, related to the conventional method, such that the difference between board numbers when a piece moved was always 4 or 5. See **Diagram 10-1.** The conventional draughts notation is given in **Diagram 10-2.**

Diagram 10-1

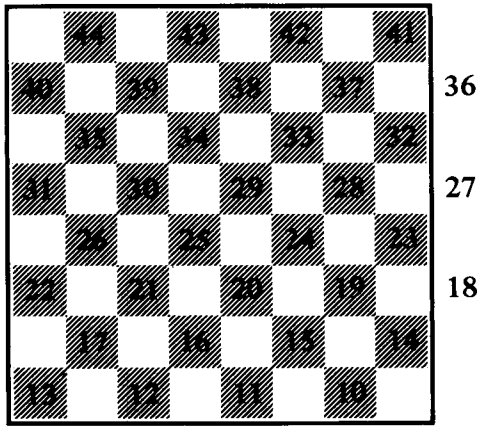
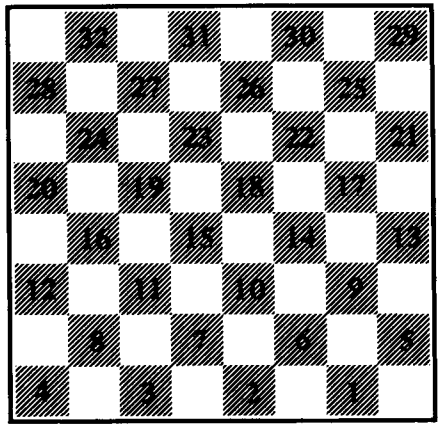


Diagram 10-2



Thus a black man on square 20 (moving up the board) will move to squares 24 or 25. A black king on square 20 will move to squares 24, 25, 16 or 15.

Note that locations 18, 27 and 36 are dummy squares, off the board. A test is needed in the move generator to tell a moving piece that a move to any square that is less than 10, greater than 44 or equal to 18, 27 or 36 is illegal.

Problem: What is the move difference when a piece captures another? What happens if you add the move difference to a piece jumping from square 23?

Several draughts programs have been published or made available in software for personal computers. Most of these combine a limited 1-ply search with a moderate evaluation function.

One of the most famous examples is that found in Creative Computing's *101 Basic Computer Games*. Simply called Checkers, it had five evaluation features, scoring high for a capture, promotion of a man to a king, moving to the side of the board and backing up one of its own pieces, and scoring low for moving a piece to where the opponent could capture it. Written in BASIC, Checkers took about five seconds to make each move.

Recently I saw a published game of draughts which contained only two features in the EF: making captures and promoting men to kings. It was claimed to play a seemingly sensible game, although I have my doubts. Numerous other versions of draughts have been published in games' collections; all seem to use a 1-ply search only.

One of the features of games of tactics – rather than of strategy – is that it is possible to obtain sensible results by assessing the value of a move rather than assessing the merit of the position which arises after the move. This can be done much faster than assessment of positions, an important factor when programming in BASIC. Creative Computing's Checkers used this approach.

A controversial problem is whether or not to give credit for, or to penalise, a move by a piece towards the side of the board. Most programs, including Checkers, favour the move to the board edge, since it reduces the danger of the piece being suddenly captured, but it misses the concept of control of the centre and Hoyle has also pointed out that a piece on the edge of the board exerts only half of its effect and has only half of its mobility. There is no easy answer to this conundrum for the programmer.

As far as I am aware, the strongest, commercially available draughts program is Borchek, which can be obtained as a cartridge for the Great Game Machine. Borchek relies almost exclusively on material evaluation coupled with a very deep search to find its moves. The only other evaluation features that I have definitely identified have been a tendency for pieces to move to the side of the board and a back-up of its own pieces.

The novice level looks at least five ply deep, while the next level takes only one second to find its moves.

Faced with a choice between weak, 1-ply draughts programs and nearly-unbeatable deep search alternatives, it does seem that there is a clear need for a shallow-search program which is hard enough to give a good game to beginners, but not so hard as to be disheartening. By far the majority of those who pore over draughts boards are casual players.

One of my own commercial programs is a draughts game called M-Checkers which has a look ahead of up to 3 ply coupled with a moderate EF which scores positions rather than moves – giving credit for backing up its own pieces, making kings, avoiding the side of the board (to improve centre control) controlling certain squares and avoiding enemy pieces.

The EF is also coupled with routines to ensure that captures must be made – no huffing is permitted – and the material count indicates whether or not one side has won. (Huffing is a rule permitted in some draughts circles, where a player refuses to make a forced capture – because the consequences look so ghastly – and loses instead the piece which should have made a capture.)

The material count also enables extra endgame routines to be called when fewer than five pieces remain. These compensate for the program's lack of look ahead in the ending. One routine is a simple proximity algorithm – the program moves its pieces towards the enemy if it is winning, away if it's losing – the other encourages movement along the two long diagonals, so that if the program is losing the piece heads for the double corner; if winning it keeps the enemy pieces out.

There is at least one draughts program commercially available for the Atari computers. This is Checker King, written in machine code and available from APX, which uses the joystick to move the pieces.

CHAPTER 11
Chess

The origin of computer chess essentially dates back to a lecture by Claude Shannon of Bell Telephone Laboratories in 1949 (published in 1950). Shannon pointed out that an average game of chess lasts forty moves for each side and that there are an average of thirty move possibilities during each of those moves. This means that there are some 10exp120 possible games of chess. To examine these at the ridiculously high rate of one million variations per second would require a search time of 10exp108 years to exhaust all possibilities.

It is certainly not possible, then, to make a computer program play a perfect game of chess by exhaustive search. If a program can choose its best move at the first ply in one second, (i.e., evaluating each position in 1/30 of a second) then, using the Minimax method of search alone, it will take 30 seconds to search to 2 ply, 900 seconds to search to 3 ply, and 27000 seconds (7.5 hours) to search to 4 ply. If it takes 1/3 second to evaluate each position, the program will take 312 days to search to a depth of 4 ply.

If the program uses optimally ordered alpha-beta pruning, taking 1/30 second per evaluation, then it will take a little over 60 seconds (allowing for time spent finding the best move at each ply) to search to 4 ply. This should make clear the advantage of using a good EF in a chess program!

Contrary to popular opinion, it is actually rather easy to write a chess program of sorts. The real difficulty is in getting it to play well at a reasonable speed. The evaluation function is likely to be the most time-consuming part of any running chess program, and in order to attain the necessary speed it will have to be written in machine code.

A good machine code program will find its best move at one ply in about one second. By comparison, the same program written in BASIC (on another computer; not the Atari) took about three minutes to find its one-ply move, while a Fortran compiler gave the same result in about five seconds.

Realistically speaking, a chess program has got to be written in machine code, although the information in this chapter may be of value to other programmers. However, a really good chess program can only be written by an expert machine code programmer, probably one who uses assembly language every day, perhaps as his daily occupation (which, incidentally,

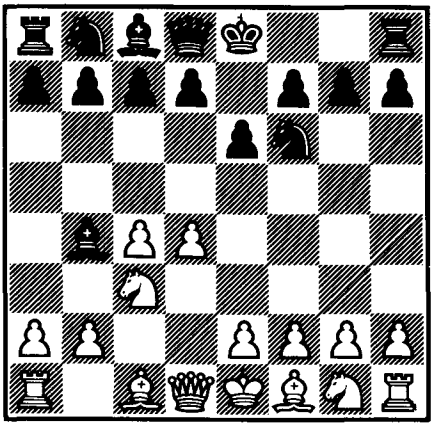
does not include me).

For a program as complex and time-consuming as chess, every available device has to be used to accelerate the program. This means optimising the code, reducing the number of page boundaries crossed and using page zero for storage of variables. Doubtless a real machine code expert has other tricks up his sleeve which I have yet to discover myself. The Atari Assembler cartridge is not very suitable for programming a chess program, since it uses much of page zero itself.

Considerable increases in speed can be accomplished simply by upgrading the central processing unit, the 6502 chip. A faster chip is available from Newell at around £30. In any case, the 6502 chip in UK Atari computers runs faster (at 2.217 MHz) than the US version at 1.79 MHz, apparently so as to compensate for the different television scan frequencies. I have no information as to whether the Newell chip is compatible with PAL television circuitry.

The general methods used to write a chess program have already been illustrated in earlier chapters of this book, including the techniques needed to set up the board, store the pieces in a piece table and search the game tree in depth, as well as the routines needed to play the opening and endgame. When setting up the board, it is necessary to consider the respective values of the bishop and the knight. Many programs set both pieces equal to 3 units (pawn = 1), but the bishop could have a value of anything up to 3.5 units. The actual figure can have a considerable influence on the effects of a pin by a bishop against a knight onto a king, such as occurs in the Nimzo-Indian defence shown in **Diagram 11-1.**

Diagram 11-1



After white has played a2–a3 in position 11-1, black can either exchange the bishop for the knight, or retreat the bishop. Which happens – assuming that no 'book' move is played from an opening library – depends

very much on the respective values of the bishop and the knight relative to the weightings involved in the change of mobility in retreating the bishop or capturing the knight.

Some of the latest commercial chess programs now dynamically alter the values of the bishop and knight according to the current position.

Chess has two moves which can only be played in special circumstances. One of these is 'en passant', using a pawn on the fifth row to capture an enemy pawn which tries to slip past it with a direct move from the seventh row to the fifth. The other is castling.

En passant is quite easy to cater for. A flag is set whenever one side makes a move enabling the other to make a capture en passant. Castling, on the other hand, is much more difficult. Not only must the program check that neither the king nor the appropriate rook have been moved previously, but it must also ensure that the king does not move through check nor settle in check. This must be done every time that castling is considered as a possible move, which has a delaying effect on the program evaluation. Early castling is not only strategically desirable by a chess program, it also speeds up the rest of the program. Do it quickly.

The difficulty involved in getting a program to castle legally can be seen from the disqualification of one commercial chess computer from the 1981 Paris Micro tournament, when it castled through check.

While on the subject of special moves, it is worth remembering that a queen move is comprised solely of a bishop and a rook, and does not need a separate move generator, and that a knight can move through other pieces (hop over them), en route to its location.

It is necessary to have a routine to test if the king is in check, both for castling and to see whether the king needs to be defended. The difficulty is to know when to call it, since it is so time consuming. Obviously, it must be called before the program makes its first move. There is no point in spending long calculation time in evaluating a position if the king must be moved.

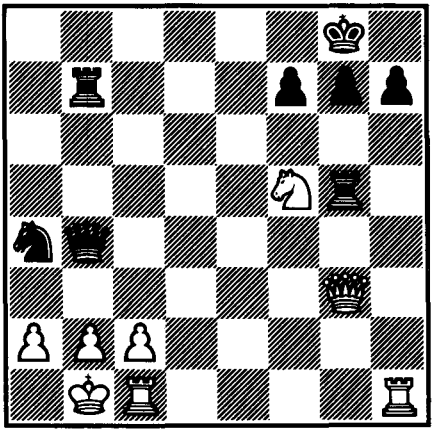
Many of the world's strongest commercial programs used to test for checkmate in every position, but the trend now is to drop this method to save time; probably the test for checkmate is done only at the lowest levels of search.

It is theoretically possible to avoid or win checkmate just by making the value of the king sufficiently high, so that capture or loss of a king automatically gains a high score. The big problem with this approach can be seen in **Diagram 11-2.**

White, to move, will permit the following sequence, thinking that the final position is favourable (+5) for it:

1. g3xg5 b4xb2
2. g3xg7 b2xb1

Diagram 11-2



3. g7xg8

In fact, of course, white lost after black's first move. It is therefore essential to test for checkmate after the first ply of search for both sides, even if you do not test at deeper plies. Remember, too, to allow for the possibility of discovered check. Just testing whether the moving piece is giving check is not sufficient.

The two most important routines in the evaluation function are the material count and the mobility assessment, a routine which measures how many squares each piece is able to move on its turn.

What else you put into the EF is up to you, but each additional feature makes the program run just that little bit slower. A chess EF is unlikely to contain more than 20 elements if it is to run quickly. Useful minor features include fianchettoing a bishop, doubling rooks along a file or column and placing rooks on the enemy seventh and eighth rows.

A more important feature to include is that of square control. When you assess each piece's mobility, you should also keep tabs on the squares which they can reach – these are influenced by the pieces which can reach them. A square is controlled if the influence of one side's pieces exceeds that of the other. The influence of a piece decreases with its value; thus a pawn has much greater influence (since it is more readily expendable) than a minor piece, and the latter has more influence than the queen.

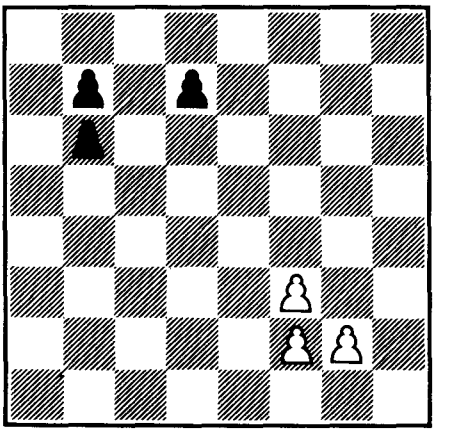
Problem: How much influence does a king have relative to a queen in the midgame? How much in the endgame?

The control of some squares, especially at the centre, is more important than that of other squares, and your EF should take this into account.

It is even possible to dispense with a lot of look-ahead investigating captures, if the EF evaluates square control properly.

The manoeuvring of pawns is complex, but the EF should certainly test for, and try to avoid, the doubling of pawns, especially if they are isolated. Isolated doubled pawns are scarcely worth more than a single pawn, whilst doubled pawns supported by others are worth perhaps 0.75 – 0.85 of a pawn each.

Diagram 11-3 Isolated and Doubled Pawns



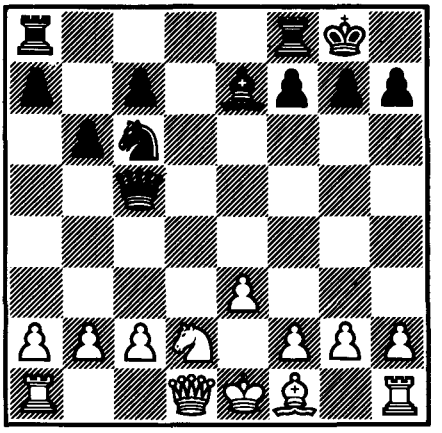
White has connected, doubled pawns.
Black has isolated, doubled pawns.

Another useful feature in the EF is to score highly any move which makes an attack on an opposing piece, particularly on the opposing king and queen. Even though deep analysis may show that this gains little, there is always the hope that the other side may blunder; in any case, chasing a piece around may finally trap it.

There are one or two commercial programs available which use 'strategic' evaluation methods, rather than purely tactical considerations. Although these methods are kept secret by the manufacturers, it does appear by analysis of the games played by their machines that their EF favours moving pieces into lines where they bear onto the enemy king; i.e., lines which are not blocked by their own pieces but may be blocked by enemy men. They also put emphasis on advancing pawn chains together. An example of a 'strategic attack' can be seen in **Diagram 11-4,** where the program would be expected to play f1–d3, h2–h4, h1–h3, h3–g3, d1–g4 in some appropriate order when the disposition of black's forces makes such moves safe.

Setting up an accurate evaluation function is tedious and mostly a matter of trial and error. It is possible to get the program to play itself, using different weightings for one feature of the EF every time it changes side, while holding the other features constant, but this will not always

Diagram 11-4



give optimal results without also changing other features of the EF. The reward for all the effort comes when the program starts to play as you originally intended.

Various methods have been used by commercial chess programmers for searching the game tree in depth. There are so many move variations in chess that some kind of selectivity is essential. Most programmers aim at a full-width search to a depth of 4-ply (even if they consider only the first few moves at this depth) which provides a good game. Some programs, as we have seen, use selective search methods with a powerful EF to pick out a handful of moves for both sides which can be examined to a depth of 5- to 6-ply. Such programs are prone to suffer from the occasional tactical blunder.

Another popular method is to use a powerful EF to score all the first ply moves for the program, and then to search deeper scoring only the material count. This works quite well, while saving a lot of computing time, but it overlooks the fact that the strategic position may have changed. Programs of this type are very prone to having doubled pawns inflicted on them, which does not affect the material count, and the opponent move, causing the program to capture with the pawn which became doubled, was not positionally evaluated.

One way round the last problem, if your program looks to a fixed depth, is to do the positional assessment again at the final depth of search. But if the program can be interrupted by a timer, or by stopping when a fixed number of positions have been evaluated, then there is no satisfactory answer.

Increasing the depth of search is subject to the law of diminishing returns, since the time required to score each deeper ply level increases geometrically. Doubling the time spent searching into the game tree rather less than doubles the strength of the play. It is much more

important to rely on a good EF for strong play.

In the 1960s, an American university program called TECH was devised. It was intended to be a 'benchmark' program against which others should be tested. Its evaluation function consisted purely of a material count; there were no other features at all, but it was able to search to considerable depth.

Against human opponents, TECH was found to be sadly deficient due to its lack of positional sense. If it could not see a powerful, material-winning tactical combination, then it just blithered and blathered. Very little time elapsed before the programmers of TECH were forced to add book openings and some positional factors in order to get a sensible game out of it.

On the other hand, chess programs are, as yet, unable to form long-range plans, and they all play each other tactically. It would be extremely interesting to know how well the original TECH would have done against some of the modern chess programs. I must emphasize very strongly, that games between chess programs do *not* necessarily reflect their performance against an inventive human opponent. Humans and computers use different methods to try to beat an opposing chess program, a point which seems to be completely misunderstood by those who play machines against each other. It does not follow that if program A beats program B, the latter graded at ELO 1700 against human opponents, that program A should be graded higher than ELO 1700, since different processes are being compared.

There is considerable debate in computing circles about whether a chess program is best advised to use a fast, full-width search or a slower, selective search to beat a human opponent.

Proponents of the first method argue that a computer program should cash in on its greatest strength, its ability to calculate fast. The extreme example of this is the current World Chess Computer Champion, the custom-designed Belle of Bell Telephone Laboratories. Belle searches deep into the game tree at colossal speed, using special chips to consider several different moves simultaneously. The main programmer claims that he does not even play chess. Belle's program is backed up by a huge 'openings and endings' library on hard disks.

The exponents of the selective approach, who include Levy and the former World Champion, grandmaster Botvinnik, counter-argue that increasing the depth of search is subject to diminishing returns and that a chess program should mimic human grandmaster play by picking out a handful of plausible alternative moves and searching these in depth to find the best one. This is known as the 'goal-plausibility' approach, and is being intensely researched on several mainframe machines.

The psychologist de Groot was able to demonstrate that human grandmasters made most of their moves on the basis of *pattern*

recognition. They recognised that any chess position was related to one that they had previously played, and were at once able to locate the decisive moves from which to select.

The problem is to program pattern recognition into a chess computer, and I suspect that advances in computer technology will support the Belle approach for some time yet, particularly if commercial programs are designed to test several moves simultaneously. However, this requires special hardware and would be unsuited to the Atari programmer.

An example of the selective approach was the old German Schach program (Shach is German for chess) which played an impressive strategic game while looking just one move ahead.

Sargon

It should be clear now that there are many new advances waiting to be made in the field of computer chess. The basic technique for writing chess programs is well-established, and there are many worthy attempts that occupy less than 4K of RAM. In fact, there is even one program which fits within a 1K Sinclair ZX-81 computer, but it uses no look-ahead and plays very badly. It shows what can be done by an expert programmer, though.

The Atari programmer does not have to worry about running out of memory, and I would strongly advise spacing out your program to begin with, not worrying too much about elegance, nor too much about speed. My first chess program occupied about 20K of RAM and played badly.

Mention must be made of *Sargon – A Chess Program* by Dan and Kathe Spracklen (Hayden Publishers). This is a book written by two of the most accomplished experts in the field, describing their first important chess program and giving the full assembly language code for SARGON-I.

The main problem with the book for an Atari user is that it is written in Z-80 code, so that it cannot be used with the Atari's 6502 microprocessor. It is a mine of useful information, though.

I wonder if the Spracklens would have published Sargon-I if they had known of the forthcoming success of the Sinclair Spectrum/Timex 2000 with its Z-80 microprocessor? I was astonished by the number of chess programs quickly released for this machine by programmers I had never heard of, all complete with en passant and castling routines nonchalantly fitted. After testing some of them, it became clear that many were blatant copies of the original Sargon-I program, adapted for the Spectrum then slightly modified and attributed to the new programmer. These programs can be distinguished by the fact that they all make the same silly moves as Sargon-I in the same silly positions.

The chess programmer can embellish his program with a few extra facilities. It can be asked to give its most favoured continuation, or a 'hint'

as to what the opponent should do, after it has completed its move calculations.

The program should always have a facility to allow the user to modify the board. This is easy to program, requiring alteration to the piece table and the main board.

Problem: Should the program be permitted to castle after modifying the board? Can the user choose for himself?

By PEEKing memory locations 18, 19 and 20, the program can operate a real-time clock. Location 20 is updated every 1/60th of a second, and the locations 18 and 19 are updated as the next higher locations reach 256. The method is outlined in Chapter 1 of this book.

The clock can be used to show the elapsed time for both sides and also to interrupt the thinking of the program and force it to display its best move found so far (using the method of iterative deepening).

In theory, it should be possible to force a machine code interrupt with the IRQ or NMI lines, but I have no personal experience of the method; not have I been able to obtain expert advice from a hardware engineer on the best means.

Finally, it is worth bearing in mind that any program capable of looking three to four ply ahead is bound to appear quite good – it will make no obvious blunders, no matter how poor its evaluation function – but a really strong chess program will require assistance with the EF from a competent chess player.

CHAPTER 12
Other Strategy Games

There are many other games of strategy which can be programmed into a computer. I have made no previous mention of the game Othello (TM in the USA), otherwise known as Reversi, for the excellent reason that I do not know the rules. However, many computer versions of Othello/Reversi exist, including two for the Atari; one from APX and another by the Spracklens from Hayden. Versions of Othello/Reversi are now so good that even the World Champion (human) has been beaten, in this case by a program called The Moor (not available for Atari computers).

The Japanese game 'GO' is so complex that it has essentially baffled attempts to synthesise a good GO program. Both Othello/Reversi and GO programs would best be written in machine code.

Turning to card games, several programs have been written to play Contract Bridge. The programming can be quite tricky, since four players need to be considered instead of the two players of most other games. Poker can also be simulated, and the computer calculates the odds rather better than most humans. For reasons that I do not understand, Cribbage has also proved to be a programmers' favourite. A 1-ply search seems to be sufficient for the computer to play an adequate game.

Backgammon is a board game that can be easily simulated in BASIC, requiring essentially only a 1-ply search. A feature of all games that rely heavily on chance is that forward searching becomes highly speculative. These games offer the richest pickings for the BASIC programmer. Ludo is very similar in principle, and Monopoly (TM) can also be programmed on the same lines.

There are two games which have drawn the attention of a number of BASIC programmers. The first of these is 3D-Noughts and Crosses which is, as the name suggests, simply an extension of ordinary noughts and crosses into three dimensions. The game is traditionally played on a board that measures 4x4x4 units and victory is achieved by the first player to complete a line of four units in any direction, including diagonally. 3D-Noughts and Crosses is much harder to play than the two dimensional version, and a good computer program will often beat a human opponent.

One problem lies in the representation of the board. Some commercial versions of the game provide elegant, multi-coloured three dimensional

pictures showing the original cube laid open like a Battenburg cake. A simpler method is to show each of four slices of the board at separate corners of the screen display. The top left slice can then be the top of the cube and the bottom right slice is the bottom of the cube.

3D-Noughts and Crosses can be satisfactorily programmed in BASIC with a 1-ply search, using much the same method as that outlined in the Noughts and Crosses program of Chapter 7. An elegant method of programming the game using 'intelligent' move selection – with random selection between moves of equal merit – was described by W N James in *Practical Computing* (January, 1981). James's program used a look-up table to assign scores to a limited range of possible moves which were all selected by reference to the last move previously played by the opponent.

James' program was written in Microsoft BASIC and occupied only 2K of memory. It could be easily converted to Atari BASIC by anyone familiar with the Microsoft dialect. It is not perfect, since the program does not consider all possible moves by the computer, only those which block the opponent's move; nevertheless, it is claimed to win 'consistently'. James made the fascinating – if immoral – proposal that the programmer alter the look-up table data so that the program will make silly moves when the programmer enters his name before commencing play. But when another player enters his name, the correct look-up table data is restored. The result is that whenever the programmer plays, he wins easily. Whenever another player tries the program, he always loses!

It should be possible to write an unbeatable program for 3D-Noughts and Crosses, still using only a 1-ply search, but it would be slow in BASIC and would need to be written in machine code to avoid long response times.

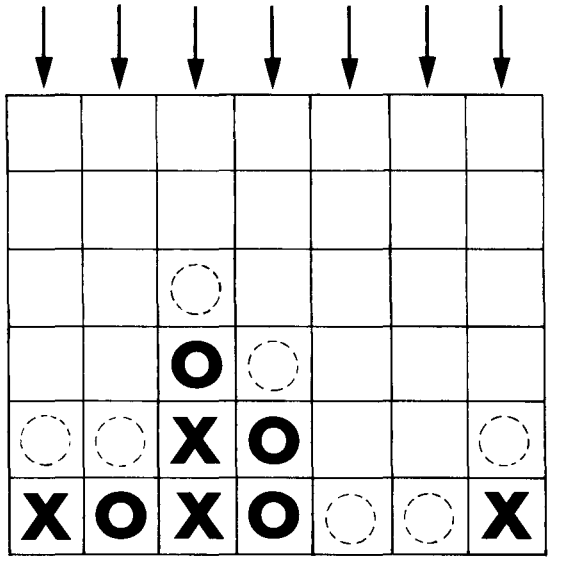
A related game is Connect-4 (TM), also known as Link-4 (TM) and Four-In-A-Row. The object of the game is for each player, taking alternative turns, to be the first to make a line of four counters in a row, vertically, horizontally or diagonally, on a board which normally measures seven squares across and six squares deep.

So far, we have only a simple, two dimensional version of 3D-Noughts and Crosses. The special characteristic of Connect-4 is that counters may only be played on the board from the bottom of the board upwards. The commercial versions of the original game are supplied with a vertical board (on a stand) into which both sides drop their counters alternately; gravity then makes the counters fall to the bottom of the board.

This means that at any one moment there are only seven possible places to put a counter: on top of any of the other counters in the seven vertical columns (see **Diagram 12-1**). When the board is nearly filled, and some of the columns are completely occupied, the number of possible moves will be less than seven.

Because the number of possible moves at each turn is so limited, a deep

Diagram 12-1



Dotted lines denote next legal moves by O (to move).

search into the game using the methods described in earlier chapters would give an invincible program. Unlike chess, the total number of moves is finite (it is actually 7exp36x6x5x4x3x2x1, roughly 2*10exp33) and although this is too large to be exhaustively searched within a reasonable period of time even by a machine code program, such a program could almost certainly 'see' further than its human opponent.

Connect-4 is also eminently suitable for programming in BASIC. The commercial versions all seem to be variants of 1-ply BASIC programs – with differing qualities – but there is no reason why the enthusiastic BASIC programmer should not write a program capable of searching to a depth of three ply.

I mentioned earlier that deep searches are difficult in Atari BASIC, since recursion – the calling of a subroutine by itself – is not permitted. This would mean that a different move generator has to be called at each level of search. Fortunately, there is a way around the problem.

The method is to generate all the legal moves at the first ply of search and store them. When all the first ply moves have been stored, each one is then separately made temporarily on the board and the opponent counter responses are made with the same move generator and again stored. Finally each of the opponent moves is temporarily made on the board and the program's counter-counter moves (at the third ply) are made with the same move generator.

The process can be carried on indefinitely, although it is rather clumsy and – in more complex games – expensive on memory. However, it should not be necessary to store all the moves at each ply of search after the first, only all the moves in response to the other side's last, temporary move.

I do encourage readers to try their hand at programming Connect-4, which is comparatively simple to design and which is still capable of providing an entertaining game. Try a 1-ply search to begin with, then a deeper search if you feel ambitious.

The Japanese game of Go-Moku requires players to form lines of five counters in a row, and is therefore conceptually related to Connect-4 and Noughts and Crosses. Go-Moku is played on a GO board of 19x19 squares, and counters can be placed anywhere on the board. As a result, Go-Moku is far more complex than the latter games, although some good machine code programs now exist to play it. These use all the techniques required of a good chess program.

Scrabble (TM) is another game which requires only a 1-ply forward search, but is complicated by the need to store a huge dictionary in hash form. One solution is to program Scrabble with a limited dictionary which the opponent is also forced to play from. A clever trick used in some commercial games to compress text is to search the text for the most commonly used clusters of letters, then to replace the clusters with tokens. A special translation routine recognises the tokens and expands them back to their original size before printing the result.

Note that many of the games I have mentioned are trademarked, and you would need a licence to offer such games for sale. There remains the possibility of inventing your own games with their own rules, designed to facilitate easy programming. An example of this is my program Warp Trog, described in Chapter 13, which was explicitly invented with a view to easy programming.

Calculation of probabilities

I have already indicated that games with an element of chance, such as Backgammon where dice are thrown, are suitable for 1-ply programs written in BASIC.

It is also possible to do forward searching in many cases, taking into account the probability that certain dice numbers will turn up. If you throw one die, the chance of any number between one and six turning up is completely even; that is, any number is equally likely to come up.

If two dice are thrown, the number which results could be any between two and twelve, but the likelihood of each number coming up is *not* even. The most likely number to be thrown is seven.

This is not the place to teach elementary statistics. Readers without

such knowledge can use **Table 12-1** to tell what the probability is that any number thrown by two dice will turn up.

Table 12-1

No. thrown by 2 dice	Probability of occurrence (%)
2	2.78
3	5.56
4	8.33
5	11.11
6	13.89
7	16.67
8	13.89
9	11.11
10	8.33
11	5.56
12	2.78

Note that alpha-beta pruning is no longer possible in game trees involving random chances.

Problem: Why not? Think about such a game tree carefully.

CHAPTER 13
Warp Trog

Warp Trog is a game of strategy designed for two players. You are one of those players, the other is your computer. The game illustrates most of the principles which have been outlined in earlier chapters.

Warp Trog is a highly sophisticated program which will play a hard game while normally taking less than fifteen seconds per move. I originally offered it for sale on another computer, but I have adapted the earlier version for this book. It is designed to play a strong, fast game, rather than to be easily understood.

The most-used routines are stacked at the head of the program which uses a 2-ply forward search to find its moves with alpha-beta pruning. The moves are sorted into score order after the first ply with a fast sort which can be seen in lines 1560-1810. The killer heuristic is found in lines 540 to 560. The chopper mechanism can be seen in line 730.

The position of all the pieces is held in a common piece table, X$, which stores the positions of both sides. The board is the array A(8,8) – using an array larger than the 6x6 board saves checking whether each move has gone off the board, when such moves are only of one square at a time.

The evaluation routines (lines 1860-2250) score each move as it is made, rather than evaluating the position which arises after the move. The former method is faster and suited to tactical games like Warp Trog.

Alphanumeric conversion is handled in lines 1820-1840. Subroutine 3090-3190 defines the character set used in the game.

A book opening library has been provided as well as extra routines to play the endgame. The library is a special, dedicated version designed to find opening moves regardless of who moves first, and is found in lines 2410-2670. The library is not related to any of the methods described in Chapter 8.

The computer will randomly select between moves of roughly equal merit. The moves are counted and error trapping routines will prevent you from making any illegal moves.

You may wish to see how effective iterative deepening is in shortening program evaluation time. First, prevent the program from sorting its moves into order by deleting line 740. Play a few games, timing the moves, and average out the result.

Next, eliminate the alpha-beta pruning by deleting line 640 and re-time the moves of a few more games. You should now realise just how effective iterative deepening is as an evaluation technique.

Read the rules before attempting to play!

The computer always plays from the top of the board. When it is your turn to move, the computer will display "YOUR MOVE".

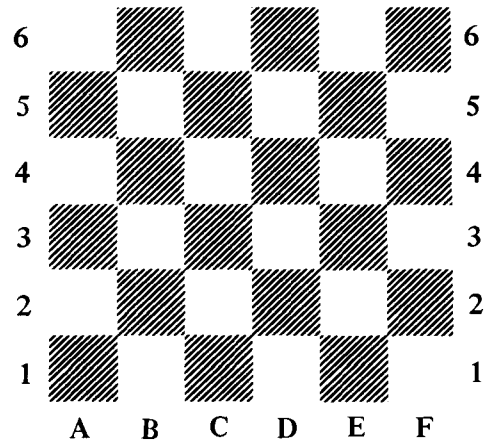
Enter Q 1 for the computer's suggestion for your move
Y 1 to enter the computer's suggestion (if desired)
W 1 to warp

Otherwise enter your move in the form X,Y where X is a letter, (a – f) and Y is a number (1 – 6), followed by X',Y' where X' and Y' have the same meanings as X and Y. There is no need to press RETURN.

The Rules

Warp Trog is played on a 6 * 6 board, labelled as follows:

Diagram 13-1



Each side has four Valgs and one Trog to begin with. The objective of the game is to prevent your opponent from moving. This can also be accomplished by taking all your opponent's pieces.

Valgs move one square at a time forwards only. On the sixth rank they are promoted to Trogs. Valgs cannot move onto a square occupied by their own side. If they move onto a square occupied by an opposing man, then that man is removed.

Trogs can move one square in any direction (except diagonally), up, down or sideways. Any enemy man on the square is removed. Trogs

cannot move onto a square occupied by their own side.

Two pieces next to each other are said to be 'supported' if they are on the same horizontal line. Supported pieces cannot be captured by enemy Valgs. They can, however, be captured separately by enemy Trogs. There is no support between pieces on the same vertical line.

Each side is permitted up to five warps. All the Trogs on that side will disappear and reappear randomly, removing any enemy pieces which they may land on. They will not land on their own pieces.

Warping is carried out from the top right square to the bottom left. Any Trog which moves left or downwards from its original position will therefore warp again – sometimes repeatedly. It is therefore desirable to move your Trogs as close to square F6 as possible before warping.

A Trog which tries to warp onto one of its own pieces will not move. If the side attempting to warp has no Trogs, then nothing will happen. This is a good way of losing a move if you wish to.

Hints for play

The closer that the warping Trog is to the top right corner of the board, the more effective will be its warping.

Although a warping Trog may hit and remove an enemy piece, it is also likely to warp onto a square where it may be easily captured. The likelihood of this depends on the number of enemy pieces and how many of them are Trogs. If you have only one Trog and your opponent has only one Trog, then warping gives you only one chance of hitting your opponent (unless you warp more than once in the same move), but there are four chances that your Trog will appear next to the enemy Trog where the enemy can capture you.

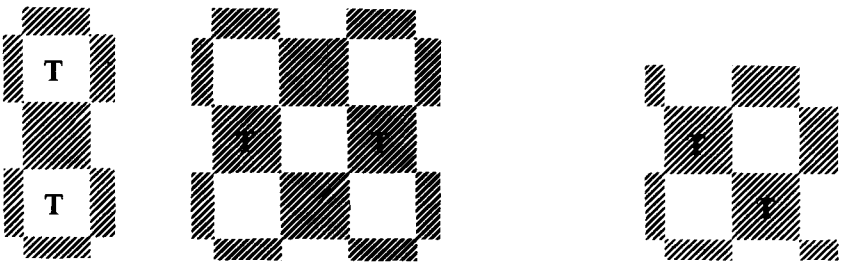
As a rule, early in the game you should close up your Valgs against enemy Valgs before warping. This reduces the chance that your Trog will appear in front of an enemy Valg where it can be captured.

Note that the side which moves first is going to have to warp first against best play by the opponent – knowing when to warp requires experience.

The concept of 'opposition' is very important in Warp Trog. Two opposing Trogs separated by one square (see **Diagram 13-2**) are said to be in 'distant opposition' . This means that the side to move has reduced mobility. If two opposing Trogs are separated by one square diagonally, then they are said to be in 'close opposition' . The side to move has greatly reduced mobility, and you should always try to place your Trogs against enemy Trogs in this way if it is then your opponent to move.

Because of 'close opposition', one side or the other is certain to win in the end, even if both sides have only one Trog left. However, remember that it is possible to warp out of trouble as long as you have

Diagram 13-2



Distant opposition **Close opposition**

some warps left.

Try to promote Valgs to Trogs as fast as possible. When you have a large number of Trogs, warping may be devastating, going on for many warps with a corresponding large number of chances of hitting your opponent.

```
90 REM ** WARP TROG **
100 SCREEN=256*PEEK(89)+PEEK(88)
110 GOTO 2800
120 FOR A=-CD TO CD STEP 2:U=I:V=J+A:GOSUB
    160:NEXT A
130 FOR A=-CD TO CD STEP 2:V=J:U=I+A:GOSUB
    160:NEXT A
140 RETURN
150 V=J-AO:U=I
160 IF (U<1) OR (U>6) THEN RETURN
170 IF (V<1) OR (V>6) THEN RETURN
180 IF SGN(A(U,V))=AO THEN RETURN
190 IF A(I,J)=CZ*AO THEN 210
200 IF (SGN(A(U,V))=-AO) AND ((SGN(A(U-CD,
    V))=-AO) OR (SGN(A(U+CD,V))=-AO)) THEN
    RETURN
210 N=N+1
220 GOSUB 1860
230 AA(N)=U:BB(N)=V:AB(N)=I:BC(N)=J
240 POSITION 0,0:PRINT #6;FLAG+1
250 Q(N)=Q
260 IF Q(N)>SO THEN SO=Q(N):S1=AA(N):S2=BB
    (N):S3=AB(N):S4=BC(N)
270 IF FLAG=0 THEN S(1)=S1:S(3)=S3:GOSUB
    1820:GOTO 290
```

```
280 GOTO 300
290 POSITION 0,0:PRINT #6;"  from ";S$(3,3);
    ",";S4;" to ";S$(1,1);",";S2
300 IF QA(K)-S0<=R(0) THEN AB=1
310 RETURN
320 DIM AA(25),BB(25),Q(25),E(20),F(20)
330 DIM C(20),D(20),AB(25),BC(25),QA(25)
340 DIM B(5,2),A(8,8),R(5),S(5)
350 DIM A$(20),R$(4),S$(4),X$(20),B$(8),C$
    (3),J$(7),A5$(20),G$(37)
360 FOR I=1 TO 6:FOR J=1 TO 6:A(I,J)=0:NEXT
    J:NEXT I
370 A(1,2)=-1:A(2,2)=-1:A(3,2)=-3:A(5,2)
    =-1:A(6,2)=-1
380 A(1,5)=1:A(2,5)=1:A(4,5)=3:A(5,5)=1:A
    (6,5)=1
390 X$="1222325262152545556 5"
400 CD=1:CZ=3
410 GOSUB 3090
420 A$="#   !   #"
430 A$(3,3)=CHR$(34):A$(5,5)=CHR$(34)
440 CU=20
450 GOSUB 2910
460 ? "DO YOU WANT TO GO FIRST? (Y/N)"
470 INPUT C$:IF C$="Y" THEN AO=1:GOSUB
    2770:GOTO 1100
480 IF C$="N" THEN GOSUB 2770:GOTO 500
490 GOTO 470
500 TO=-100:MO=MO+CD:R(0)=-100
510 AO=1:SO=-99:AB=0:FLAG=0:PA=1
520 IF BO=0 THEN GOSUB 2410:IF BO=0 THEN 960
530 GOTO 580
540 I=S(3):J=S(4):SO=-99:AB=0:U=S(1):V=S(2)
550 IF K=1 THEN 630
560 ON A(I,J)+4 GOSUB 210,210,200
570 GOTO 630
580 FOR J1=1 TO 5
590 I=VAL(X$(J1*2+9,J1*2+9)):J=VAL(X$(J1*2+
    10,J1*2+10))
600 ON A(I,J)+4 GOSUB 140,140,140,140,150,
    150,120,120
610 NEXT J1
620 GOTO 680
630 FOR JK=1 TO 5
```

```
640 IF AB=1 THEN 670
650 I=VAL(X$(JK*2-1,JK*2-1)):J=VAL(X$
    (JK*2,JK*2))
660 ON A(I,J)+4 GOSUB 120,120,150,140,140,
    140,140
670 NEXT JK
680 IF N<>0 THEN 710
690 IF FLAG=0 THEN POSITION 7,2:PRINT #6;
    "YOU WIN ":? :GOTO 2750
700 IF FLAG=1 THEN POSITION 8,2:PRINT #6;
    "I WIN ":GOTO 2710
710 IF FLAG=1 THEN RETURN
720 M=N:FLAG=1:A0=-1:PA=6
730 IF M=1 THEN GOSUB 2780:GOTO 930
740 GOSUB 1560
750 R(0)=-99
760 FOR K=1 TO M
770 E(M-K+1)=AA(K):F(M-K+1)=BB(K):C(M-K+1)
    =AB(K):D(M-K+1)=BC(K)
780 QA(M-K+1)=Q(K):Q(K)=0:AA(K)=0:BB(K)=0:
    BC(K)=0:AB(K)=0
790 NEXT K
800 FOR K=1 TO M
810 N=0
820 RY=A(E(K),F(K)):A(E(K),F(K))=A(C(K),
    D(K)):A(C(K),D(K))=0
830 GOSUB 540
840 QZ=QA(K)-S0
850 IF QZ>R(0) THEN 870
860 GOTO 910
870 R(0)=QZ:R(1)=E(K):R(2)=F(K):R(3)=C(K):
    R(4)=D(K)
880 S(1)=S1:S(2)=S2:S(3)=S3:S(4)=S4
890 GOSUB 1820
900 POSITION 0,0:PRINT #6;"  from ";R$(3,3)
    ;",";R(4);" to ";R$(1,1);",";R(2)
910 A(C(K),D(K))=A(E(K),F(K)):A(E(K),F(K))
    =RY
920 NEXT K
930 IF R(0)<-0.9 AND R(0)>-7 AND WA<6 THEN
    WA=WA+1:GOSUB 2290:GOTO 970
940 IF RND(1)>0.2 OR R(0)>0.1 THEN 960
950 IF (QT<7 OR QT>12) AND QY<-6 AND WA<6
    THEN WA=WA+1:GOSUB 2290:GOTO 970
```

```
960 GOSUB 1820:GOTO 1000
970 S$="WARP":S(2)=0:S(4)=0
980 POSITION 0,0:PRINT #6;" WARP "
990 GOTO 1100
1000 POSITION 0,0:PRINT #6;"  from ";R$(3,3)
     ;",";R(4);" to ";R$(1,1);",";R(2)
1010 IF R(2)=1 AND A(R(3),R(4))=1 THEN A(R
     (3),R(4))=3
1020 IF ABS(R(1)-R(3))=1 AND R(2)=3 AND
     A2=R(1) AND B2=R(2)+1 THEN A(A2,B2)=0
1030 RY=A(R(1),R(2)):A(R(1),R(2))=A(R(3),
     R(4)):A(R(3),R(4))=0:IF B0=0 THEN 1080
1040 FOR I=1 TO 5:IF X$(2*I+9,2*I+10)=STR$
     (R(3)*10+R(4)) THEN X$(2*I+9,2*I+10)=S
     TR$(R(1)*10+R(2)):QF=I
1050 NEXT I
1060 FOR I=1 TO 5:IF X$(2*I-1,2*I)=X$(2*QF
     +9,2*QF+10) THEN X$(2*I-1,2*I)="00"
1070 NEXT I
1080 POSITION R(1)*2+2,(7-R(2))*2+2:? #6;
     CHR$(ASC(A$(A(R(1),R(2))+4,A(R(1),R(2))
     +4))-32)
1090 POSITION R(3)*2+2,(7-R(4))*2+2:?
     #6;A$(4,4)
1100 FLAG=0:N=0:T0=-100:A0=CD
1110 SOUND 0,80,10,6:FOR L=1 TO 40:NEXT L:
     SOUND 0,0,0,0
1120 POSITION 0,5:? "MOVE = ";M0
1130 A5$="YOUR MOVE"
1140 POSITION 0,7:PRINT A5$
1150 QT=0:QY=0:FOR I=1 TO 5:I1=A(VAL(X$
     (I*2-1,I*2-1)),VAL(X$(I*2,I*2)))
1160 J1=A(VAL(X$(I*2+9,I*2+9)),VAL(X$(I*2+
     10,I*2+10))):QT=QT+I1:QY=QY+J1:NEXT I
1170 IF QT=0 THEN POSITION 7,2:PRINT #6;
     "YOU WIN ":GOTO 2750
1180 IF QY=0 THEN POSITION 8,2:PRINT #6;
     "I WIN ":GOTO 2750
1190 TRAP 3200:GOSUB 3020
1200 A1=LETTER
1210 POKE SCREEN+CU*40+20,A1-32
1220 GOSUB 3020
1230 B1=LETTER
```

```
1240 POKE SCREEN+CU*40+21,B1-32
1250 IF A1=89 AND S(2)<>0 THEN A1=S(3)+64:
     B1=S(4)+48:A2=(S1)+64:B2=S(2)+48:GOTO
     1350
1260 IF A1=87 AND WB<6 THEN WB=WB+1:GOSUB
     2290:GOTO 1540
1270 IF A1=87 AND WB>5 THEN A5$="NO WARPS":
     GOTO 1140
1280 IF A1=81 THEN PRINT "TRY ";S$(3,3);S(4)
     ;"  ";S$(1,1);S(2);"  ":GOTO 1190
1290 GOSUB 3020
1300 A2=LETTER
1310 POKE SCREEN+CU*40+25,A2-32
1320 GOSUB 3020
1330 B2=LETTER
1340 POKE SCREEN+CU*40+26,B2-32
1350 A1=A1-64:A2=A2-64:B1=B1-48:B2=B2-48
1360 IF A(A2,B2)<0 THEN 1450
1370 IF A(A1,B1)>-1 THEN 1450
1380 IF A(A1,B1)=-CD THEN 1420
1390 IF ABS(B2-B1)=1 AND ABS(A2-A1)=0
     THEN 1460
1400 IF ABS(B2-B1)=0 AND ABS(A2-A1)=1
     THEN 1460
1410 GOTO 1450
1420 IF B2-B1<>CD OR A2<>A1 THEN 1450
1430 IF A(A2,B2)>0 AND (A(A2+CD,B2)>0 OR
     A(A2-CD,B2)>0) THEN 1450
1440 GOTO 1460
1450 GOSUB 2260:GOTO 1140
1460 IF B2=6 AND A(A1,B1)=-1 THEN A(A1,B1)
     =-3
1470 POSITION A2*2+2,(7-B2)*2+2:PRINT #6;
     CHR$(ASC(A$(A(A1,B1)+4,A(A1,B1)+4))+96)
1480 POSITION A1*2+2,(7-B1)*2+2:PRINT
     #6;A$(4,4)
1490 A(A2,B2)=A(A1,B1):A(A1,B1)=0
1500 QZ=0:FOR I=1 TO 5:IF X$(2*I-1,2*I)=STR$
     (A1*10+B1) THEN X$(2*I-1,2*I)=STR$(A
     2*10+B2):QZ=2*I-1
1510 NEXT I
1520 FOR I=1 TO 5:IF X$(2*I+9,2*I+10)=X$(QZ,
     QZ+1) THEN X$(2*I+9,2*I+10)="00"
```

```
1530 NEXT I
1540 PRINT "                "
1550 GOTO 500
1560 SS=1:POSITION 0,0:PRINT #6;"#"
1570 B(1,1)=1:B(1,2)=M
1580 LL=B(SS,1):RR=B(SS,2):SS=SS-1
1590 II=LL:JJ=RR:XX=Q(INT(RND(1)*(RR-LL)
     +0.5)+LL)
1600 IF Q(II)>=XX THEN 1620
1610 II=II+1:GOTO 1600
1620 IF XX>=Q(JJ) THEN 1640
1630 JJ=JJ-1:GOTO 1620
1640 IF II>JJ THEN 1710
1650 WW=Q(II):Q(II)=Q(JJ):Q(JJ)=WW
1660 WW=AA(II):AA(II)=AA(JJ):AA(JJ)=WW
1670 WW=BB(II):BB(II)=BB(JJ):BB(JJ)=WW
1680 WW=AB(II):AB(II)=AB(JJ):AB(JJ)=WW
1690 WW=BC(II):BC(II)=BC(JJ):BC(JJ)=WW
1700 II=II+1:JJ=JJ-1
1710 IF II<=JJ THEN 1600
1720 IF JJ-LL>=RR-II THEN 1760
1730 IF II>=RR THEN 1750
1740 SS=SS+1:B(SS,1)=II:B(SS,2)=RR
1750 RR=JJ:GOTO 1790
1760 IF LL>=JJ THEN 1780
1770 SS=SS+1:B(SS,1)=LL:B(SS,2)=JJ
1780 LL=II
1790 IF LL<RR THEN 1590
1800 IF SS>0 THEN 1580
1810 RETURN
1820 FOR KI=1 TO 3 STEP 2
1830 R$(KI,KI)=CHR$(ASC(STR$(R(KI)))+16):
     S$(KI,KI)=CHR$(ASC(STR$(S(KI)))+16)
1840 NEXT KI
1850 RETURN
1860 Q=RND(1)/40
1870 Q=Q+ABS(A(U,V))
1880 IF A(U,V-A0)=-CZ*A0 THEN Q=Q-2
1890 IF A(U,V-A0)=-A0 THEN Q=Q-0.75:
     GOTO 1910
1900 GOTO 1920
1910 IF (SGN(A(U+CD,V))=A0 OR SGN(A(U-CD,V))
     =A0) AND ABS(J-V)=1 THEN Q=Q+0.75
```

```
1920 IF A(U,V+AO)=-CZ*AO THEN Q=Q-0.6*ABS
     (A(I,J))
1930 IF A(U-CD,V)=-CZ*AO OR A(U+CD,V)=-CZ*AO
     THEN Q=Q-0.6*ABS(A(I,J))
1940 IF A(I,J+AO)=-CZ*AO OR A(I-CD,J)=-CZ*AO
     OR A(I+CD,J)=-CZ*AO THEN Q=Q+0.35
1950 IF A(I,J)=AO THEN 2160
1960 Q=Q+U/100+V/100
1970 IF QT-QY<10 THEN Q=Q+(1/(ABS(U-A2)+8))
     +(1/(ABS(V-B2)+8))
1980 IF (I=3 OR I=4) AND (J=3 OR J=4) THEN
     Q=Q-0.1
1990 Q=Q-((U=1)+(U=6)+(V=1)+(V=6))/18
2000 Q=Q+((I=1)+(I=6)+(J=1)+(J=6))/10
2010 IF A(U,V+AO)=-AO THEN Q=Q+0.3
2020 FOR CA=-2 TO 2 STEP 4
2030 IF U+CA<CD THEN NEXT CA
2040 IF SGN(A(U+CA,V))=-AO THEN Q=Q+0.15*
     ABS(A(U+CA,V))/2
2050 IF V+CA<CD THEN NEXT CA
2060 IF SGN(A(U,V+CA))=-AO THEN Q=Q+0.15*
     ABS(A(U,V+CA))/2
2070 NEXT CA
2080 FOR CA=-1 TO 1 STEP 2
2090 IF (A(I+CA,J-CA)=-CZ*AO) OR (A(I+CA,
     J+CA)=-CZ*AO) THEN Q=Q-0.22
2100 IF SGN(A(U+CA,V-CA))=-AO THEN Q=Q+0+0.
     1*ABS(A(U+CA,V-CA))
2110 IF SGN(A(U+CA,V+CA))=-AO THEN Q=Q+0.
     1+0.1*ABS(A(U+CA,V+CA))
2120 IF A(U+CA,V)=-AO THEN Q=Q+0.3
2130 IF A(U+CA,V)=AO THEN Q=Q+0.1
2140 NEXT CA
2150 RETURN
2160 IF V=PA THEN Q=Q+0.5
2170 IF V=PA+AO THEN Q=Q+0.3
2180 IF V=PA+2*AO THEN Q=Q+0.2
2190 IF SGN(A(U-CD,V))=AO THEN Q=Q+0.2
2200 IF SGN(A(U+CD,V))=AO THEN Q=Q+0.2
2210 QX=0:FOR CA=V TO PA STEP -AO
2220 QX=QX+A(U,CA)
2230 NEXT CA
2240 IF SGN(QX)=0 OR SGN(QX)=AO THEN
```

```
     Q=Q+0.15
2250 RETURN
2260 A5$="ILLEGAL  ":SETCOLOR 4,3,10:SOUND
     0,120,10,6:FOR L=1 TO 40:NEXT L
2270 SOUND 0,0,0,0:SETCOLOR 4,0,0
2280 RETURN
2290 FOR I=6 TO 1 STEP -1:FOR J=6 TO 1
     STEP -1
2300 SETCOLOR 0,I,J
2310 SOUND 1,5*I,10,6:SOUND 2,5*J,10,6
2320 QY=INT(6*RND(1)+1):QZ=INT(6*RND(1)+1)
2330 IF A(I,J)=-CZ*A0 THEN 2350
2340 GOTO 2370
2350 IF SGN(A(QY,QZ))=A0 THEN SOUND 0,4,4,6:
     FOR L=1 TO 100:NEXT L:SOUND 0,0,0,0
2360 IF SGN(A(QY,QZ))<>-A0 THEN A(QY,QZ)=A
     (I,J):A(I,J)=0:GOSUB 2840
2370 NEXT J:NEXT I
2380 SETCOLOR 0,2,8
2390 SOUND 1,0,0,0:SOUND 2,0,0,0
2400 GOSUB 2930:B0=1:RETURN
2410 IF M0=1 THEN J$="4445344":GOTO 2680
2420 IF M0=2 AND A(4,4)=CZ AND A(4,2)<>-CZ
     AND A(3,3)<>-CZ THEN 2440
2430 GOTO 2460
2440 IF A(3,4)=-CZ THEN J$="3,4,4,4,3,34":
     GOTO 2680
2450 J$="4344343":GOTO 2680
2460 IF M0=2 THEN QZ=A2
2470 IF QZ=0 THEN B0=1:RETURN
2480 ON QZ GOTO 2490,2530,2560,2590,2620,
     2650
2490 IF M0=2 THEN J$="2425224":GOTO 2680
2500 IF M0=3 AND A2=2 AND B2=3 THEN J$=
     "1415114":GOTO 2680
2510 IF M0=3 AND A2=3 AND B2=3 THEN J$=
     "5455454":GOTO 2680
2520 B0=1:RETURN
2530 IF M0=2 THEN J$="1415114":GOTO 2680
2540 IF M0=3 AND A2=1 AND B2=3 THEN J$=
     "2425224":GOTO 2680
2550 B0=1:RETURN
2560 IF M0=2 THEN QZ=INT(RND(1)*2):IF
```

```
     A(6-QZ,3)=0 THEN 2580
2570 BO=1:RETURN
2580 R(1)=6-QZ:R(2)=4:R(3)=6-QZ:R(4)=5:X$((
     5-QZ)*2+9,(5-QZ)*2+10)=STR$(10*R(1)+R
     (2)):RETURN
2590 IF MO=2 AND A(4,5)=CZ THEN J$="4445344"
     :GOTO 2680
2600 IF MO=2 AND A(4,4)=CZ AND A(6,3)=0
     THEN J$="6465564":GOTO 2680
2610 BO=1:RETURN
2620 IF MO=2 AND A(6,3)=0 THEN J$="6465564"
     :GOTO 2680
2630 IF MO=3 AND A(5,3)=-CD THEN J$=
     "5455454":GOTO 2680
2640 BO=1:RETURN
2650 IF MO=2 THEN J$="1415114":GOTO 2680
2660 IF MO=3 AND A(1,3)=-CD THEN J$=
     "5455454":GOTO 2680
2670 BO=1:RETURN
2680 FOR I=1 TO 5:R(I)=VAL(J$(I,1)):NEXT I
2690 X$(R(5)*2+9,R(5)*2+10)=J$(6,7)
2700 RETURN
2710 R(3)=C(K):R(1)=E(K):GOSUB 1820:R(2)=
     F(K):R(4)=D(K)
2720 POSITION 0,0:PRINT #6;"  from ";R$(3,3)
     ;",";R(4);" to ";R$(1,1);",";R(2)
2730 POSITION R(1)*2+2,(7-R(2))*2+2:? #6;
     CHR$(ASC(A$(A(R(1),R(2))+4,A(R(1),R(2))
     +4))-32)
2740 POSITION R(3)*2+2,(7-R(4))*2+2:PRINT
     #6;A$(4,4)
2750 SOUND 0,60,10,6:SOUND 1,150,10,6:FOR
     K=1 TO 100:NEXT K:SOUND 0,0,0,0:SOUND
     1,0,0,0
2760 ? "  ":END
2770 PRINT "                                ":RETURN
2780 R(0)=S0:R(1)=S1:R(2)=S2:R(3)=S3:R(4)=S4
2790 S(1)=0:S(2)=0:S(3)=0:S(4)=0:RETURN
2800 GRAPHICS 17:POKE 752,1
2810 POSITION 5,5:? #6;"WARP TROS"
2820 POSITION 4,8:? #6;"initialising"
2830 GOTO 320
2840 ON AO+2 GOTO 2880,2850,2850
```

```
2850 FOR K=1 TO 5:IF X$(2*K+9,2*K+10)=STR$
     (QY*10+QZ) THEN X$(2*K+9,2*K+10)="00"
2860 IF X$(2*K-1,2*K)=STR$(10*I+J) THEN X$
     (2*K-1,2*K)=STR$(10*QY+QZ)
2870 NEXT K:RETURN
2880 FOR K=1 TO 5:IF X$(2*K-1,2*K)=STR$(QY*
     10+QZ) THEN X$(2*K-1,2*K)="00"
2890 IF X$(2*K+9,2*K+10)=STR$(10*I+J) THEN
     X$(2*K+9,2*K+10)=STR$(10*QY+QZ)
2900 NEXT K:RETURN
2910 GRAPHICS 1:POKE 752,1
2920 POKE 756,F/256
2930 FOR I=1 TO 6:FOR J=6 TO 1 STEP -1
2940 POSITION 16,J*2+2:? #6;7-J
2950 POSITION I*2+2,(7-J)*2+2
2960 W=A(I,J):IF W<0 THEN ? #6;CHR$(ASC
     (A$(W+4,W+4))+96)
2970 IF W>0 THEN ? #6;CHR$(ASC(A$(W+4,W+4
     ))-32)
2980 IF W=0 THEN ? #6;A$(W+4,W+4)
2990 NEXT J:NEXT I
3000 POSITION 4,16:? #6;"A B C D E F"
3010 RETURN
3020 CLOSE #1:OPEN #1,4,0,"K:"
3030 GET #1,LETTER
3040 IF (LETTER>48) AND (LETTER<55)
     THEN 3080
3050 IF (LETTER>64) AND (LETTER<71)
     THEN 3080
3060 IF (LETTER=81) OR (LETTER=87) OR
     (LETTER=89) THEN 3080
3070 GOTO 3020
3080 RETURN
3090 FOR I=1 TO 36:READ A:POKE ADR(G$)+I,A:
     NEXT I
3100 F=(PEEK(106)-8)*256
3110 DATA 104,104,133,205,104,133,204,104,
     133,207,104,133,206,104,104,133,208,16
     6,208,160,0
3120 DATA 177,204,145,206,200,208,249,230,
     205,230,207,202,208,240,96
3130 Q=USR(ADR(G$)+1,224*256,F,4)
3140 FOR I=0 TO 31:READ B:POKE F+I,B:NEXT I
```

```
3150 DATA 0,0,0,0,0,0,0,0
3160 DATA 255,129,129,129,129,129,129,255
3170 DATA 255,255,255,255,255,255,255,255
3180 DATA 24,60,126,255,255,126,60,24
3190 RETURN
3200 TRAP 40000:GOSUB 2260:GOTO 1140
```

CHAPTER 14
Proceed with Caution

Any program must be thoroughly tested before use, as well as being debugged. Obviously, any program must be debugged before it can be used at all and this section deals only with the more subtle problems which may arise.

It is essential that every line of the program is tested at some stage, or you may get a surprise when a rarely used routine – such as en passant in chess – gets its first airing. Most of the program can probably be tested in blocks, or subroutines, but check off every line against your master listing, then GOTO or GOSUB any lines or blocks which remain. It is likely that you will collect some error codes in this way even if no genuine error exists, so make allowances for ERROR 13 (NEXT without FOR) or ERROR 16 (RETURN without GOSUB) and similar problems.

It is also absolutely essential to check that your evaluation function is giving the results that you intended. Set up a few standard positions and evaluate them; compare the result with that which you have calculated by hand (first delete any randomising routines).

The piece table, if you have one, is another potential trouble area. Check that when a piece is moved on the board (or off it, by capture), the piece table is updated. Otherwise pieces will become 'lost' on the board because they are not referenced by the piece table.

Another common problem is with the screen display. Make sure that your pieces move correctly on the screen, with particular care taken in the opening (if a book library is provided) and on the final move. I have often forgotten to make the winning side complete its final move before announcing that it has won. A similar problem arises when the program operates the 'Chopper' mechanism – when it is forced to make its one and only legal move at once – again remember to update the piece table and the screen display.

Error messages such as 'Illegal Move' must be catered for without permitting them to scroll onto other parts of the screen. The Atari computers' split screen modes are very useful for such messages. The display can be kept intact in the upper part of the screen, while the lower part scrolls all input data and error messages out of sight.

This brings us to the dreaded error-trapping of the user's input, long

the bane of my life. All the user's input, usually of moves, must be checked for validity – it is sometimes possible to couple the move generator to the input routine to test that the move is a legal one, but it is usually easier to write a special input routine.

The Atari's TRAP statement can be used to filter out certain types of error, such as typing a letter instead of a number, or vice-versa, or dealing with a number that is too large for an array, but it cannot check the legality of moves.

I always find that it takes me many, many attempts to write a complete error checking routine which eliminates all user mistakes; it is my experience that almost as much time gets spent on this part of the program as on any of the more difficult of the other sections.

It is very good discipline always to write any program as though you will later offer it for sale, even if you do not actually intend to do so. The Atari Program Exchange offers a very useful booklet to anyone who hopes to submit a program to APX for publishing; it contains numerous hints on how to give your program a professional appearance.

This book outlines all the general principles that you will need to write a game of strategy and I have given several examples to aid comprehension; nevertheless, it is worth emphasising that the principles will need to be adapted to each and every program. I have often used the principle underlying the book opening program given in Chapter 8 in my own strategic games, but have had to modify the actual running routines – as well as the book library itself – every time that I have used it to date.

I also seem to need to change the operation of the alpha-beta pruning mechanism at repeated intervals, depending on whether the EF is called after every move, after every position, or is only called in full at the lower levels of search.

Writing any strategy game requires a great deal of thought before you start programming. You should consider carefully what each part of the program is trying to accomplish, and whether a faster or more elegant routine could be written. I have shuddered at some of the things I have programmed in the past, and at some future date I shall very likely shudder at parts of the ideas and concepts described in this book. The opening library in Warp Trog does not follow the principles expounded in Chapter 8, but I have left it alone for the sound reason that it works as well as I would wish. As a general rule, if something works well for you, then leave it alone!

Endgame

If you have followed this book through carefully, and worked out all the examples, you should be in a position to write your own intelligent games.

The day may come when you are the proud author of a complicated and powerful program to play a strategy game – it may be chess, it may be something else – and you will want to set about finding out just how strong it really is.

The most important point to remember is that your program will be played by a human, not by another computer, and so it is against other humans that it must be assessed.

Most programs employ completely different processes to find their moves, to that used by a human player. Games between computers are therefore a very poor test of how well your computer program would perform against an inventive human opponent.

Always test your programs against human opponents, not against other computers.

INDEX

(Numbers refer to chapters where topic is principally dealt with).

652

Writing Strategy Games on Your Atari®

Techniques for Intelligent Games

John White

Your opponent is a machine. It reacts with unnerving intelligence. Your position seems lost, but you rally and "out-think" your computer for a brilliant win!

That's the excitement of playing against your Atari! Now you can create your own strategy games, such as chess, for hours of fun and excitement.

Writing Strategy Games on Your Atari explores key programming techniques and gives you concrete examples to help you build your own strategy games.

Critical techniques are fully explained and illustrated. For example, you'll learn how to:

- Monitor scoring and evaluate players' positions
- Program the computer to make its best move
- Set up a game board
- Generate moves
- Increase the speed and efficiency of a program, and much more!

Strategy games like GO, bridge, Connect-4, and Othello are also discussed. Includes games ready for you to key in, play, modify, and enjoy!

This book is written for intermediate and advanced BASIC programmers. Tips on how to modify the routines are also provided for BASIC and machine code programmers.

HAYDEN BOOK COMPANY
a division of Hayden Publishing Company, Inc.
Hasbrouck Heights, New Jersey

ISBN 0-8104-6528-0