Chapter 9 Difficult board games − the beginnings of Artificial Intelligence

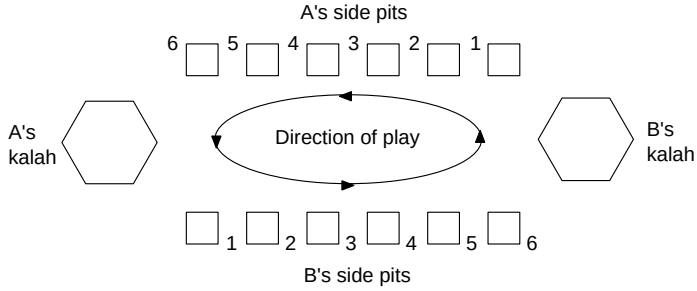
In the last chapter, the games that we used as examples were all fairly easy games with moderately sized game trees. In this chapter, we are going to introduce the methods that are needed for programming more difficult games. We shall demonstrate these methods by using an ancient game called Kalah. Kalah is somewhat easier than Chess and Draughts, but is still very much more intellectually demanding than any of the games that have been used so far. It is ideal for experimentation with game playing techniques.

9.1 The game Kalah

We now present the rules for Kalah and suggest that you familiarise yourself with these rules by finding scaneone who will play a few games with you. (You can mark out a board on a sheet of paper and use coins or counters for stones.)

The Rules of Kalah

Kalah Originated as a desert game played by two people using stones and holes made in the sand. We can imagine the game to be played on a rectangular board:



In front of him, each player has six pits numbered 1 to 6 (called his 'side' pits). To the right of his side pits, each player has a special pit called a 'kalah'. Initially, all the side pits contain an equal number of stones. A move consists of taking the stones from one of one's own side

pits and distributing them anti-clockwise one by one into the other pits including one's own kalah but not the opponent's. There are two rules:

(a) Players make moves alternately except when the last stone of a pile that a player moves lands in his own kalah. That player then makes another move.

(b) If the last stone of a pile that a player is moving lands in an empty pit on his own side, that stone along with any stones in the pit opposite are placed in that player's kalah.

The game ends when the player whose turn it is cannot make a move (ie. all his side pits are empty). Each person's score is the number of stones in his kalah plus the number (if any) in his side pits. The player with the highest score is the winner.

For a good beginner's game, start with 2 or 3 stones in each side pit. Starting with 6 stones in each pit results in quite a difficult garne.

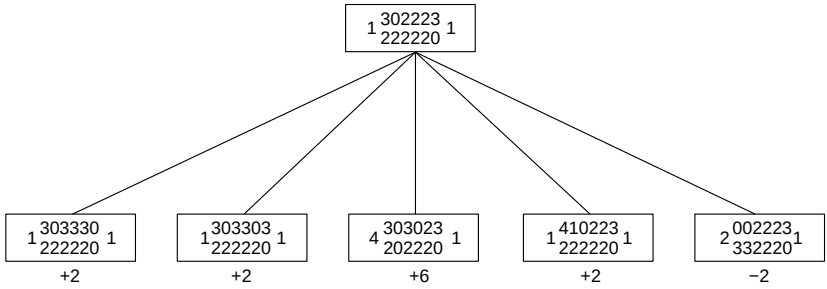
9.2 Static evaluation functions

If you have run the recursive procedures for exploring the 'Last One Wins - or does he?' game tree for positions with 9 or 10 counters, you will have discovered that, even for a trivial game like this, the computer takes a fairly long time to do an exhaustive exploration of the game tree. For games like Kalah, Draughts or Chess, exhaustive exploration of the game tree is completely impossible.

The first possible alternative that we consider is to make a program select a move by carrying out a one-level lookahead from its current position. In order to compare the moves available in a given position, we need some way of comparing the new positions reached by the available moves. Since we have ruled out the possibility of an exhaustive minimax evaluation of these positions, we need some alternative evaluation mechanism. One possibility is to define a 'static evaluation function' which makes some numerical estimate of the goodness of a position without further exploration of moves and counter-moves. The precise definition of the static evaluation function will of course depend on the rules of the game.

In Kalah, we could calculate a static value (from A's point of view) for a board position by adding up the stones in A's pits together with his kalah and subtracting the stones in B's pits and kalah. This simple function could be improved upon, but it will suffice for our present purposes.

The next diagram! illustrates the process of choosing a move by using a one-level lookahead together with a static evaluation function.



The program will try each move available, apply its static evaluation function to each position reached and choose the move that leads to the position with the best static value.

9.3 An introductory Kalah program

Before discussing improvements to the simple approach described in the last section, we shall write a complete Kalah program that uses a one-level lookahead to choose its moves.

We use as our framework the procedure PROCplaygame defined in Chapter 1. Before filling in the details by writing the other procedures required, we must decide how te represent a Kalah board in our program. There are various possibilities available. One that immediately suggests itself is a 7x2 array of integers where each integer represents the number of stones in one of the pits. Another possibility would be to pack a board position into two numeric variables and represent the number of stones in one pit by one hexadecimal digit. Such a representation would not be very convenient from the point of view of making moves, but it might be useful if we ever wanted to store large numbers of board positions in a table and needed to economise on storage space.

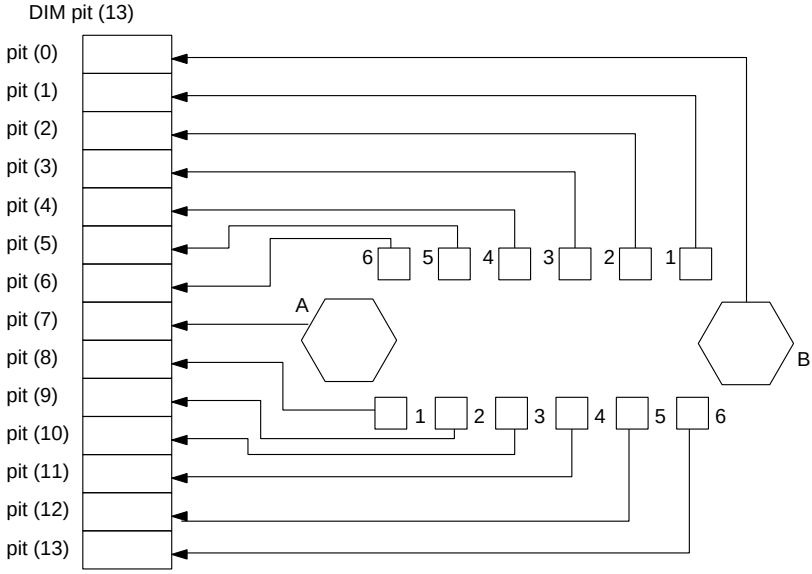
We shall in fact represent the Kalah board using a one-dimensional array of 14 locations numbered 0 to 13:

```
    5  DIM pit(l3)
```

Although this does not reflect the two-sided structure of the board, it is an extremely convenient representation in which to manipulate moves. For the purpose of making moves, the Kalah board is circular, and moving round the board corresponds to increasing the subscript of our one-dimensional array. We can use the MODoperator to make sure that the subscript goes back to zero if we reach 13:

```
        pitno = (pitno + 1) MOD 14
```

The representation chosen is illustrated:



We first define a procedure for displaying a board position that is represented in this way. For the time being we simply print the number of stones in a pit as a number on the screen. Whenever a move has been made, this procedure will be used to display a completely new representation of the board. We leave as an exercise the use of character graphics to produce a more realistic display of the board.

```
205   DEF PROCdisplayboard
210   LOCAL p
215     @% = &0303
220     PRINT:PRINT:PRINT "     (6)(5)(4)(3)(2)(1)"
225     PRINT TAB(3);
230     FOR p= 6 TO I STEP -1 : PRINT pit(p); : NEXT p
235     PRINT
240     PRINT pit(7); TAB(21); pit(0)
245     PRINT TAB(3);
250     FOR p= 8 TO 13 : PRINT pit(p); : NEXT p
255     PRINT : PRINT "     (1)(2)(3)(4)(5)(6)" : PRINT
260     @% = &0A0A
265   ENDPROC
```

The board would be initialised by the following procedure:

```
300  DEF PROCsetupboard
310  LOCAL p, stones
320    INPUT "How many stones per pit", stones
330    FOR p = 1 TO 6
340      pit(p) = stones
350    NEXT p
360    FOR p= 8 TO 13
370      pit(p) = stones
380    NEXT p
390    pit(0) =0 : pit(7) = 0
400    INPUT "Do you want to start", reply$
410    IF LEFT$(reply$,1) = "Y" THEN turn$="B"
         ELSE turn$="A"
420    ENDPROC
```

The next two procedures are straightforward:

```
500  DEF PROCtestgameover
510  LOCAL moves, start, p
520    IF turn$="A" THEN start = 1
         ELSE start = 8
530    moves = 0
540    FOR p= start TO start+5
550      IF pit(p) > 0 THEN moves = moves+1
560    NEXT p
570    gameover = (moves=o)
580  ENDPROC

700  DEF PROCannouncewinner
710  LOCAL Apoints, Bpoints, p
720    Apoints = pit(7) : Bpoints = pit(0)
730    FOR p=1 TO  6 : Apoints=Apoints+pit(p) : NEXT
740    FOR p=8 TO 13 : Bpoints=Bpoints+pit(p) : NEXT

750    IF Apoints>Bpoints THEN PRINT "I win!"
         ELSE IF Bpoints>Apoints THEN PRINT "You win!"
         ELSE PRINT "Its a draw."

760    PRINT "Final score -  ME:"; Apoints
770    PRINT "              YOU:"; Bpoints
780  ENDPROC
```

There are a number of useful relationships that we can take advantage of in manipulating board positions. For example, if 'k' is the location of one player's kalah (k=0 or k=7), then '7-k' gives the location of the other player's kalah; if 'p' is the location of one of the pits then '14-p' is the location of the opposite pit.

We now define the procedures for inputting and making the program's opponent's moves.

```
 900  DEF PROCplayerB
 910  LOCAL pitno
 920    PROCinputmove
 930    PROCmakeBmove(7+pitno)
 940  ENDPROC

1000  DEF inputmove
1010    PRINT"Your move, which pit do you play from";
1020    INPUT pitno
1030    IF FNlegaBmove(pitno) THEN ENDPROC

1040    REPEAT
1050      PRINT "Illegal Move. "
1060      INPUT "Try again:" pitno
1070    UNTIL FNlegalBmove(pitno)
1080  ENDPROC

1100  DEF FNlegalBmove(p)
1110    IF p<0 OR p>7 THEN =FALSE
1120    IF pit(7+p)<=0 THEN =FALSE
1130  =TRUE
```

It is slightly more convenient to define two different procedures for makingmoves, one for player A and one for player B. The two procedures will be almost identical, but defining them separately eliminates some testing. The move-making; procedures must decide whose turn it is next, setting the value of 'turn$' accordingly.

```
1200  DEF PROCmakeAmove(p)
1210  LOCAL lastpit
1220    PROCmovestones(p ,0)
1230    IF lastpit<7 AND pit(lastpit)=1 THEN
          capture = pit(14-lastpit) :
          pit(14-lastpit) =0 : pit(lastpit) =0 :
          pit(7) = pit(7)+capture+1
        ELSE capture = -1 :REM signifies no capture
1240    IF lastpit = 7 THEN turn$="A" ELSE turn$="B"
1250  ENDPROC

1400  DEF PR0CmakeBmove(p)
1410  LOCAL lastpit
1420    PROCmovestones(p,7)
1430    IF lastpit>7 AND pit(lastpit)=1 THEN
          capture = pit(14-lastpit) :
          pit(14-lastpit) = 0 : pit(lastpit) =0 :
          pit(0) = pit(0)+capture+1
        ELSE capture = -1
1440    IF lastpit = 0 THEN turn$="B" ELSE turn$="A"
1450  ENDPROC
```

```
1600  DEF PROCmovestones(p, oppkalah)
1610  LOCAL s
1620    a = pit( p) : pit(p) = 0
1630    REPEAT
1640      p = (p+1) MOD 14
1650      IF p<>oppkalah THEN pit(p)=pit(p)+1 : s=s-1
1660    UNTIL s=0
1670    lastpit = p
1680  ENDPROC
```

We now define the procedures that carry out the one-level lookahead from a position at which it is the program's turn to play.

```
1800  DEF PROCplayerA
1810  LOCAL move
1820    move = FNbestmove
1830    PRINT "I move from pit "; move
1840    PROCmakeAmove(move)
1850  ENDPROC

2000  DEF FNbestmove
2010    LOCAL p, maxsoiar, bestmove
2020      maxsofar = -100
2030      FOR p = 1 TO 6
2040        IF pit(p)>0 THEN PROCtryAmove
2050    NEXT p
2060  =bestmove

2200  DEF PROCtryAmove
2210  LOCAL stones, v, capture
2220    stones = pit(p)
2230    PROCmakeAmove(p)
2240    v = FNstaticval
2250    IF v>ymaxsofar THEN maxsofar=v : bestmove=p
2260    PROCmoveback(p, stones, capture, 0)
2270  ENDPROC
```

Note the need to move the stones back to their original positions each time a move has been tried. This is done by PROCmoveback which needs to be told at which pit the move started, how many stones were in the pit before the move and how many stones, if any, were captured. At the moment, this procedure is used only for undoing a move by player A, but we shall shortly use it for undoing a move by B and it will be convenient if we give the procedure a 4th parameter telling it the location of the opponent's kalah.

```
2400  DEF PROCmoveback(p, a, capt, oppkalah)
2410    pit(p) = pit(p) + a
2420    REPEAT
2430      p = (p+1) HOD 14
2440    IF p<>oppkalah THEN
          pit(p) = pit(p) - 1 : a = s-1
2450    UNTIL s=0
2460    IF capt>-1 THEN
          pit(p)=pit(p)+1 : pit(14-p)=pit(14-p)+capt :
          pit(7 -oppkalah) = pit(7-oppkalah)-capt-1
2470  ENDPROC
```

Finally, we define FNstatieval which is straightforward:

```
2600  DEF FNstaticval
2610  LOCAL p, points
2620    points = pit(7) - pit(0)
2630    FOR p= 1 TO  6 : points=points+pit(p) : NEXT
2640    FOR p= 8 TO 13 : points=points-pit(p) : NEXT
2650  = points
```

The program that we have defined so far plays an extremely poor game of Kalah. You will find that it is quite easily beaten. In the next few sections we shall look at ways of improving the program's performance. We shall do this by redefining FNbestmove and PROCtryAmove in various ways, leaving the rest of the program unchanged.

Exercises

1 Change FNbestmove so that it considers moves in reverse order, ie. using:

```
FOR p = 6 TO 1 STEP -1
```

Why does this improve the program's performance for games starting with two or three counters?

2 Try to improve the program's performance by defining an improved static evaluation function. For example, if a large pile of stones builds up in pit 6, then these eventually have to be distributed into one's opponent's pits. The program could be discouraged from doing this by making the static evaluation function give less credit for stones in pit 6. The credit given for stones in other pits could be varied in the same sort of way.

3 Modify the Kalah progrtun so that it does not display a fresh copy of the board each time a move is made. It should instead use the TAB function to ujxiate the

existing display changing the numbers already on the screen in the same order as the pits; are changed while making a move. Use time delays so that the numbers do not change too quickly. If you are familiar with the use of Teletext colour and large character facilities in MODE 7, you could make use of these to enhance the display.

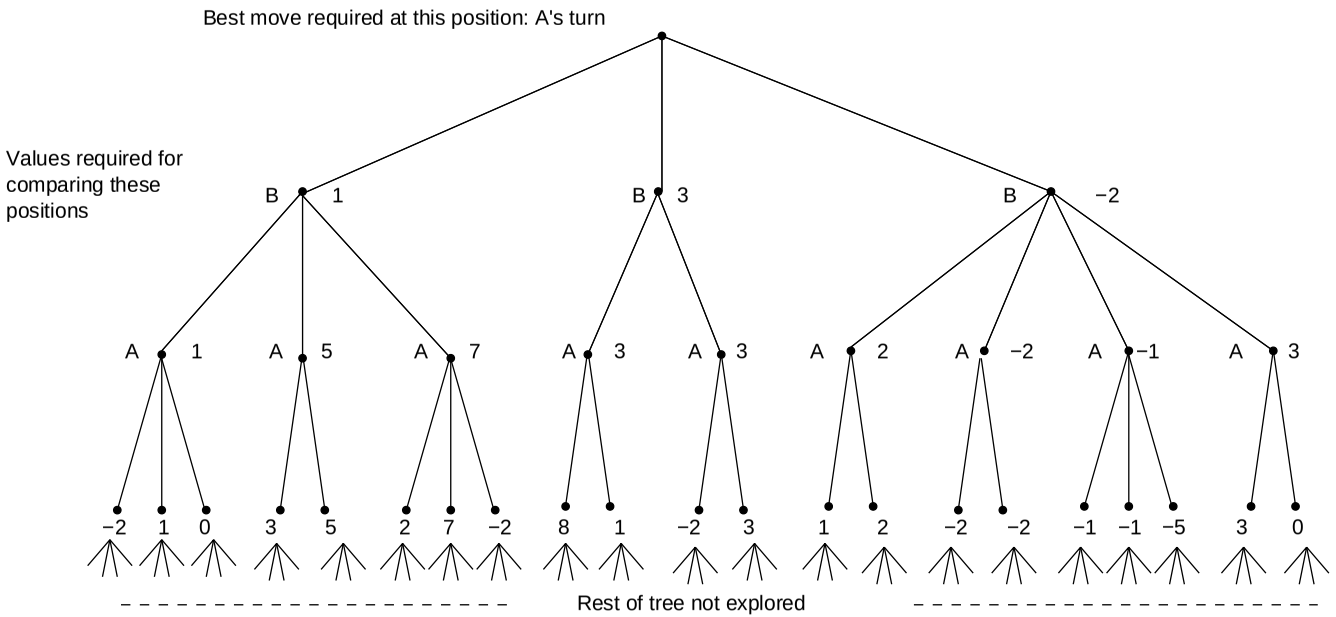
9.4 Looking further ahead in non-trivial games

In the last section, we saw how a simple static evaluation function could be used by a program for evaluating and comparing positions. The program that we produced did not play very well. Its performance could be improved by defining a better static evaluation function, but you will find that the extent to which it can be improved is limited. This is because it is difficult for a static evaluation function to take account of the possible sequences of moves and counter-moves that are available in the position being evaluated. The availability of certain moves can make the true value of a position very different from the value obtained by a static evaluation function. For example, the program might apply a static evaluation to a position in which its opponent was about to make a spectacular capture. In such circumstances, the value obtained will clearly be misleading.

We have already decided that exhaustive exploration of the game tree is out of the question, but a possible compromise way of evaluating a position is to look ahead in the game tree as far as time and space allow, perform a static evaluation of the positions reached, and minimax with these values in order to obtain a value for the starting position. A static value is calculated only after a sequence of moves and counter-moves has been tried. The value obtained by minimaxing with static values obtained in this way will hopefully be a better estimate of the value of the initial position than would have been obtained by using the static evaluation function alone. The minimax value of a position is the static value of some position that can be reached from the first position by a 'likely' sequence of moves and counter-moves.

One way of limiting the extent of the lookahead would be to impose some bound on the depth of the subtree that is to be explored by the program.

This is illustrated in the next diagram for a hypothetical game. The root of the subtree is a position at which the program has to decide on a move. To do this we have to obtain values for each of the three positions to which moves are available. These values are obtained by looking ahead to a further depth of 2 and minimaxing with the static values of the positions reached. Thus the program chooses a move by carrying a lookahead with an overall depth of 3.


Best move required at this position: A's turn
Values required for comparing these positions
B 1
B 3
B −2
A 1
A 5
A 7
A 3
A 3
A 2
A −2
A −1
A 3
−2 1 0
3 5
2 7 −2
8 1
−2 3
1 2
−2 −2
−1 −1 −5
3 0
Rest of tree not explored

The structure of the procedure required to carry out a minimax lookahead of the type just described will be essentially the same as the structure of the procedures that carried out an exhaustive minimax lookahead for 'Last One Wins - or does he?'. We shall write two functions, FNminval for obtaining the value of a position at which it is B's turn and FNmaxval for a position at which it is A's turn.

In 'Last One Wins - or does he?', lookahead was terminated when a terminal pasition of the game was reached. In this case, lookahead will be terminated if a 'depth bound' is reached (or earlier if a terminal position of the game is reached). Each of our two minimax functions needs to be given a depth parameter that can be used to test whether or not to terminate the lookahead.

Before we define FNminval and FNmaxval, let us define a modified version of PROCtryAmove that can be used by FNbestmove.

```
 2200  DEF PROCtryAmove
 2210  LOCAL stones, v, capture, turn$
 2220    stones = pit(p)
 2230    PROCmakeAmove(p)
 2240    IF turn$="B" THEN v=FNminval(1)
           ELSE v=FNmaxval(1)
 2250    IF v>maxsofarTHEN maxsofar=v : bestmove=p
 2260    PROCmoveback(p, stones, capture, 0)
 2270  ENDPROC
```

There are two points to note here. We have made use of the fact that PROCmakeAmove sets 'turn$' to indicate whose turn it is in the new position created. This information is used to decide whether we use FNminval or FNmaxval to evaluate the new position. The parameter, 1, given to FNminval or FNmaxval indicates that the position to be evaluated is at depth 1 from the position at which a move has to be made. We shall assume in what follows that a variable 'maxdepth' has been set to an appropriate value, say:

```
    6  maxdepth = 2
```

FNminval and FNmaxval are defined as:

```
 3000  DEF FNminval(depth)
 3010  LOCALminsofar, p
 3020    IF depth=maxdepth THEN=FNstaticval
 3030    minsofar=100
 3040      FOR p = 8 TO 13
 3050        IF pit(p)>0 THEN PROCtestBmove
 3060    NEXT p
 3070  IF minsofar<100 THEN =minsofar ELSE =FNstaticval
```

```
 3100  DEF PROCtestBmove
 3110  LOCAL stones, v, capture, turn$
 3120    stones = pit(p)
 3130    PROCmakeBnove(p)
 3140    IF turn$="A" THEN v=FNmaxval(depth+1)
           ELSE v=FNminval(depth+1)
 3150    IF v<minsofar THEN minsofar=v
 3160    PROCmoveback(p,stones,capture,7)
 3170  ENDPROC

 3200  DEF FNmaxval(depth)
 3210  LOCAL maxsofar, p
 3220    IF depth=maxdepth THEN=FNstaticval
 3230    maxsofar=100
 3240    FOR p = 1 TO 6
 3250      IF pit(p)>0 THENPROCtestAmove
 3260    NEXT p
 3270  IF maxsofar>-100 THEN =maxsofar ELSE =FNstaticval

 3300  DEF PROCtestAmove
 3310  LOCAL stones, v, capture, turn$
 3320    stones = pit(p)
 3330    PROCmakeAmove(p)
 3340    IF turn$="B" THEN v=FNminval(depth+1)
           ELSE v=FNmaxval(depth+1)
 3350    IF v>maxsofar THEN maxsofar=v
 3360    PROCmoveback(p,stones,capture,0)
 3370  ENDPROC
```

Our two minimax evaluation functions, together with their two subsidiary procedures PROCtestBmove and PROCtestAmove, look rather more complicated than their counterparts for 'Last One Wins - or does he?'. However, you should be able to see that they work in essentially the same way. An activation of FNminval tests its depth and terminates it 'maxdepth' has been reached. If the maximum depth has not been reached then FNminval tries each pit in turn and, if a move is available from that pit, it is made (by PROCtestBmove) and the resulting position is evaluated by calling FNmaxval or FNminval recursively with the depth parameter increased by 1 (line 3140). Each move that is tried is always undone by PROCmoveback.

There is a possibility that FNminval will be called for a position in which no moves are available. If this happens, then all the pits will be examined (line 3050) with no moves being tried and no recursion taking place. This possibility is covered by the IF-statement at line 3070. If 'minsofar' has not been set as a result of trying a move, this means that no moves are available and the position must be evaluated by FNstaticval.

FNmaxval and its subsidiary procedure PROCtestAmove are identical in structure to FNminval and PROCtestBmove. You

will notice that PROCtestAmove is the same as PROCtryAmove except that PROCtryAmove records 'bestmove' as well as 'maxsofar'. PROCtryAmove could have been used instead of PROCtestAmove, but we have used two different procedures to draw attention to the different contexts in which they are used.

If you run this version of the program, you will find that, with 'maxdepth' set to 3, it can take 30 seconds or more to choose a move. With 'maxdepth' set to 4, it can take five or six times as long. We will be looking at ways of improving these times in the next section. However, even with 'maxdepth' set as low as 2, the performance of the program is considerably better than that of the version that used only a one-level lookahead. It now tries not to leave stones where they can be captured on the next me and it can spot situations where gaining an extra turn gives it an immediate capture move (for example, moving from pit 5 at the start of the game).

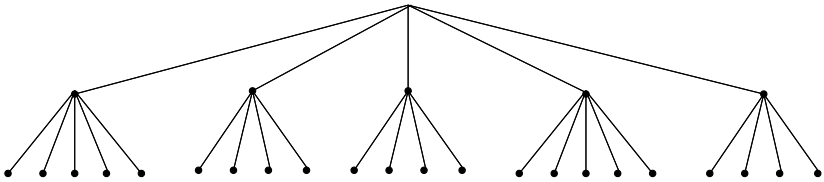
Now that we have presented the structure of our minimax lookahead procedures, it will be fairly easy to make various improvements to these procedures. For example, the main test as to whether the lookahead should terminate appears at the head of FNminval and FNmaxval. This test can be easily modified if we wish to alter the conditions for terminating the lookahead. Stopping at a fixed depth and carrying out a static evaluation is almost as arbitrary as stopping at level one. A position at our maxinnam depth might be part of a capture-recapture sequence along which the static values fluctuate wildly. For this reason most game-playing programs use a variable-depth lookahead. They attempt to terminate the lookahead in 'stable' situations, even if this means looking ahead further than the maximum depth along some branches of the tree.

In our Kalah program, we might decide to terminate the lookahead only if both the maximun depth has been reached and the last move was not a capture. We can arrange this by changing the test at the head of each of FNminval and FNmaxval as:

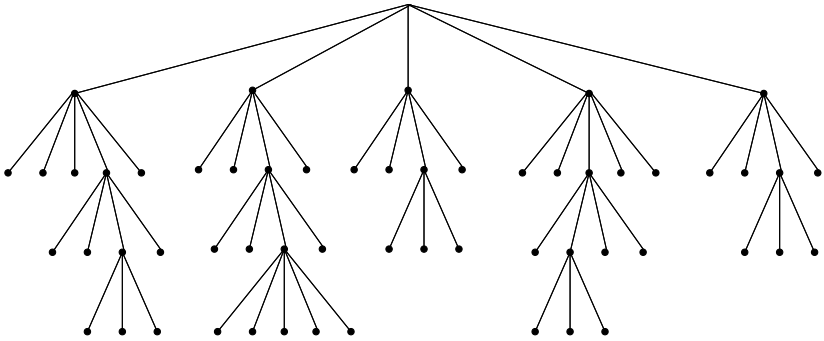
```
3020   IF depth>=maxdepth AND capture<=0 THEN =FNstaticval

3220   IF depth>=maxdepth AND capture<=0 THEN =FNstaticval
```

Here, the structure of a typical lookahead tree for Kalah using this more flexible stopping rule is compared with that of a tree in which the lookahead stops at a fixed depth bound.



Typical structure of tree explored with maxdepth = 2



Typical structure of search tree explored with maxdept = 2
and capture-recapture sequences being explored further

It might be necessary in some games to impose a second maximum depth bound beyond which this exploration of unstable sequences has to stop. We could do this by adding a second stopping test at the head of our minimaxing functions, for example:

```
3025   IF depth=2*maxdepth THEN =FNstaticval

3225   IF depth=2*maxdepth THEN =FNstaticval
```

Exercises

1 A game-playing program often allows the user to ask the computer for help. Change the program so that the user can ask it to recommend a move. To do this you will have to extend the definition of PROCinputmove so that it uses a new function, FNbestBmove.

2 Alter PROCplayerA so that it times the call of FNbestmove and reports the exact time taken by the program to choose its move. This alteration will be useful in the next exercise, and later for comparing the effectiveness of

various tree-pruning technique.

3 In some applications, the speed of execution of part of a program may be critical, for example in processing real-time data or in animation. In such circumstances the programmer may be justified in attempting to speed up execution of the relevant sections of program by shortening variable, function and procedure names, by using '%' variables where possible, and even using GOTO andGOSUB statements where appropriate. By using such tricks, see if you can increase the speed at which the program selects a move. Note that there is no point in attempting to speed up procedures such as PROCplaygame, PROCsetupboard, PROCdisplayboard, etc. Procedures that are obeyed only once per game once per move are hardly critical. Concentrate on FNminval, FNmaxval and any functions and procedures used by them. Obtain some timings for the fast program and compare these with the timings for the original version. Are the time savings worth the decrease in readability of the program?

Note that on the BBC computer, BASIC programs are normally interpreted when we RUN them. In Chapter 10, we discuss the difference between an interpreter and a compiler. Perhaps we should state here that if our game-playing program were being compiled before being run, then changes like those described above would have little or no effect on the execution time of the program.

9.5 Tree pruning

We have now described the overall approach that is used by programs that play board games like Chess, Draughts and Kalah. We shall devote the remainder of this chapter to techniques for improving the structure of the lookahead tree explored by such programs when choosing a move.

It is an fact of life that difficult games have very 'bushy' game trees. For example, Chess has an average of about 30 moves available in each position. Each time we go down one level in the Chess tree we find that there are 30 times as many positions as there were at the previous level. We say that the 'branching factor' of the Chess tree is 30.

If a Chess program looks two moves ahead it could examine 1+30+900 positions in the process. An 'obvious' way of improving the quality of the program's play is to make it look further ahead when deciding on its move. If the program were to look 4 moves ahead, it could exantine 1+30+900+27000+810000 positions which would take nearly 900 times as long.

In our Kalah playing program, looking 3 moves ahead can take half a minute or more. As the program stands, increasing 'maxdepth' would make it unusable.

Increasing the depth of a program's lookahead would be

quite impossible without the help of 'tree-pruning' techniques. These allow the program to ignore certain branches of its lookahead tree and use the time saved to explore other branches more deeply.

Alpha beta pruning - an intoduction
No game-playing program can afford not to use the technique known as 'alpha-beta pruning'.

Some of the pruning methods that we mention later involve a risk that the program will ignore a move that is superficially bad but which would have turned to be the best move available. For example, moves in Chess that that sacrifice material in exchange for a superior position can be missed as a result of over enthusiastic tree pruning.

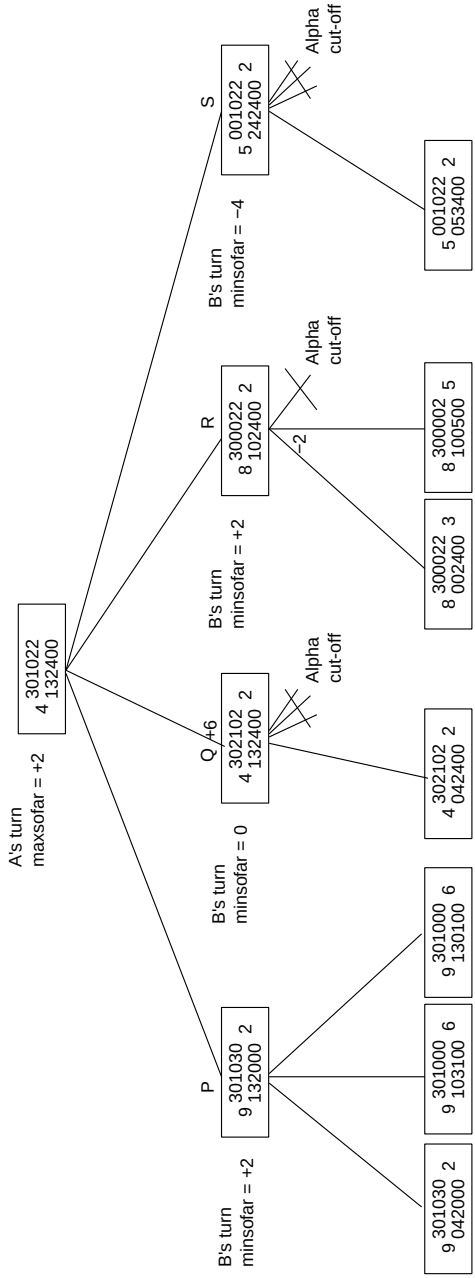
There is no such risk associated with alpha-beta pruning. Exactly the same move will be chosen by the program whether or not the technique is used. If it is used, it can result in spectacular reductions in the number of positions examined during a lookahead.

We introduce alpha-beta pruning by considering two very simple examples of its use. In the next diagram, we illustrate a depth 2 lookahead from the Kalah position at the root of the subtree. For convenience we shall refer to the 4 positions at depth 1 as positions P, Q, R and S. We assume that a program is conducting a minimax search from left to right as we have drawn the tree.

Position P is considered first. After the program has considered all the moves available in position P, the value of 'minsofar' is known to be +2 and this value is returned to the level above as the final value for wsition P. At this stage 'maxsofar' at the root of the tree is +2.

The program now examines position Q. The first move considered in position Q leads to a position with a static value of 0 and, when this position has been examined, 'minsofar' for position Q is 0. The 'minsofar' value at position Q can only get smaller and we can see at this stage that, even if we consider all the other moves available in position Q, the final value of position Q must be less than or equal to 0 which is in turn less than the value of 'maxsofar' at the root of the tree. Whatever the final value of position Q, it cannot possibly be greater than the current value of 'maxsofar' at the root. The value obtained for position Q can therefore have no effect on the value obtained for the position at the root or on the move chosen at the root. It would thus be a waste of time considering the other moves that are available in position Q. This is an example of an 'alpha cut-off'. (We shall see where the term 'alpha' comes from in a moment.)

A similar thing happens in positions R and S. As soon as the value of 'minsofar' becomes less than or equal to the value of 'maxsofar' at the position inntediately above, an alpha cut-off takes place and the program can stop trying moves at the current position.


A's turn
maxsofar = +2
4 301022
4 132400
P
B's turn
minsofar = +2
9 301030 2
9 132000
B's turn
minsofar = +2
9 301030 2
9 042000
+8
9 301000 6
9 103100
+2
B's turn
minsofar = 0
9 301000 6
9 130100
+2
Q +6
4 302102 2
4 132400
Alpha
cut-off
4 302102 2
4 042400
0
B's turn
minsofar = +2
R
8 300022 2
8 102400
Alpha
cut-off
−2
8 300022 3
8 002400
+6
8 300002 5
8 100500
+2
B's turn
minsofar = −4
S
5 001022 2
5 242400
Alpha
cut-off
5 001022 2
5 053400
−4

In order to see a simple example of a 'beta cut-off', we have to examine a lookahead of depth 3 or more. The diagram on the next page illustrates part of a fixed depth lookahead tree with 'maxdepth' set to 3.

During its attempt to obtain a value for position T (at which it is player B's turn) the program explores the subtree shown. Position U has a value of +4 and this beccunes the value of 'minsofar' at position T. Position V is then examined and the first move available at position V leads to a value of +6. Position V now has a 'maxsofar' of +6 and considering other values at position V can only cause this value to get bigger. The final value obtained for position V cannot affect the value of 'minsofar' at position T above. A beta cut-off takes place and no other moves at position V need be tried. A similar cut-off takes place at position W.

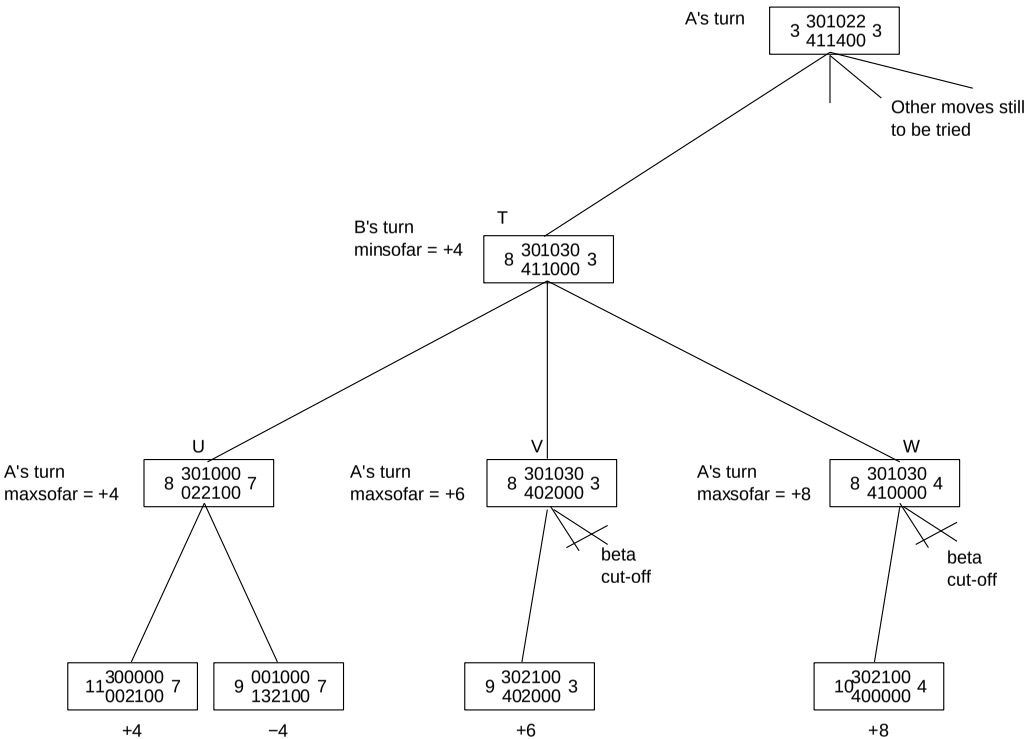
Alpha-beta pruning in its complete form is rather more general than has been indicated here. Before we complicate matters by giving a full description, it will be useful to implement the method as it has been described so far.

We start by considering the so-called alpha cut-off that ocurred at position Q in the last but one diagram. This is a position at which it is B's turn to play and it is evaluated by FNminval. The main loop in this function tries each possible move in turn:

```
3040  FOR p = 8 TO 13
3050  IF pit(p)>0 THEN PROCtestBmove
3060  NEXT p
```

We now have to cater for the possibility that this loop has to terminate prematurely (because of an alpha cut-off) and it must therefore be replaced by a REPEAT loop. (No GOTOs jumping out of FOR-loops please!) The loop terminates either because all possible moves have been tried or because 'minsofar' has become less than or equal to the current maxsofar' value of a node above. This 'maxsofar' value must be passed into FNminval as a parameter. Because this value comes £rem a position at which it is A's turn, it has traditionally been called an 'alpha value' (hence the term alpha cut-off'). FNminval has to be modified as follows:

```
3000  DEF FNtminval(depth, alpha)
3010  LOCAL minsofar, p
3020    IF depth=maxdepth THEN =FNstaticval
3030    minsofar=l00
3040    p = 7
3050    REPEAT
3060      p = p+1
3070      IF pit(p)>0 THEN PROCtestBmove
3080    UNTIL p=13 OR minsofar<=alpha
3090  IF minsofar<100 THEN minsofar ELSE =FNstaticval
```


A's turn
3 301022 411400 3
Other moves still to be tried
B's turn
minsofar = +4
T
8 301030 411000 3
U
A's turn
maxsofar = +4
8 301000 022100 7
V
A's turn
maxsofar = +6
8 301030 402000 3
beta cut-off
W
A's turn
maxsofar = +8
8 301030 410000 4
beta cut-off
11 300000 002100 7
+4
9 001000 132100 7
−4
9 302100 402000 3
+6
10 302100 400000 4
+8

The variable-depth alteration could of course be included at line 3020.

A similar alteration can be made, to FNmaxval to allow for the possibility of a beta cut-off, where beta is the 'minsofar' value of a position above at which it is B's turn.

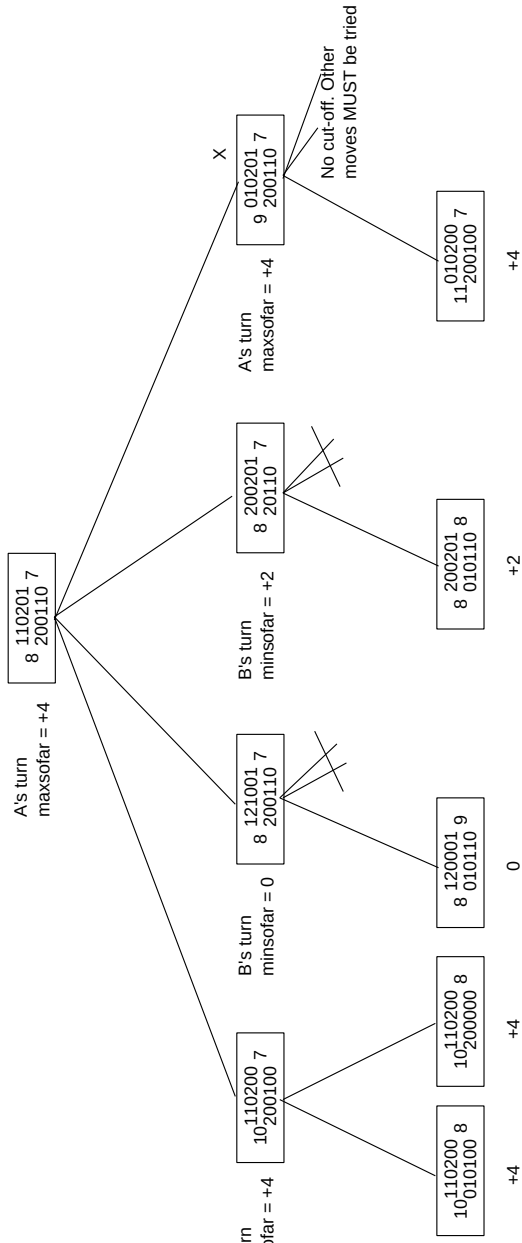
```
3200  DEF FNmaxval(depth, beta)
3210  LOCAL maxsofar, p
3220    IF depth=maxdepth THEN =FNstaticval
3230    maxsofar=100
3240    p=0
3250    REPEAT
3260      p = p+1
3270      IF pit(p)>0 THEN PROCtestAmove
3280    UNTIL p=6 OR maxsofar>=beta
3290  IF maxsofar>-100 THEN =maxsofar ELSE =FNstaticval
```

We must now modify the procedures that call FNminval and FNmaxval and ensure that an appropriate alpha or beta value is always supplied as a parameter. A slight complication occurs in the situation where the program tries a move that entitles the player whose turn it is to another turn. Pruning can take place only as a result of comparing a 'minsofar' value with a 'maxsofar' value or vice versa. We must take care not to supply a 'minsofar' value to a call of FNminval or a 'maxsofar' value to a call of of FNmaxval. PROCtryAmove tries a move at the root of the subtree being explored and when it has made a trial move it needs to call FNminval or Flimaxval as follows:

```
2240    IF turn$="B" THEN v=FNminval(1,maxsofar)
          ELSE v=FNmaxval(1,100)
```

The beta value of 100 given to FNmaxval simply indicates that there is no 'minsofar' value above the position to be evaluated. No pruning can take place at a position to be evaluated by FNmaxval in this context. A situation in which this occurs is illustrated on the next page.

The fourth move tried at the start position of the lookahead tree is one that entitles player A to another turn. The position marked 'X' therefore has a 'maxsofar' value associated with it. At the stage reached in the lookahead, this 'maxsofar' value is equal to the 'maxsofar' value at the root of the tree, but this does not cause pruning to take place. The 'maxsofar' value at position X may easily increase as a result of trying other moves at position X and any increase in this value will eventually cause a change in the 'maxsofar' value at the root of the tree.


8 110201 7
200110
A's turn
maxsofar = +4
B's turn
minsofar = +4
10 110200 7
200100
10 110200 8
010100
+4
10 110200 8
200000
+4
B's turn
minsofar = 0
8 121001 7
200110
8 120001 9
010110
0
B's turn
minsofar = +2
8 200201 7
20110
8 200201 8
010110
+2
X
A's turn
maxsofar = +4
9 010201 7
200110
No cut-off. Other
moves MUST be tried
11 010200 7
200100
+4

The two subsidiary procodures, PROCtestBmove and PROCtestAmove, that are used by FNminval and FNmaxval, also need to be changed. In PROCtestBmove we require

```
3140    IF turn$="A" THEN v=FNmaxval(depth+1, minsofar)
          ELSE v=FNminval(depth+1, alpha)
```

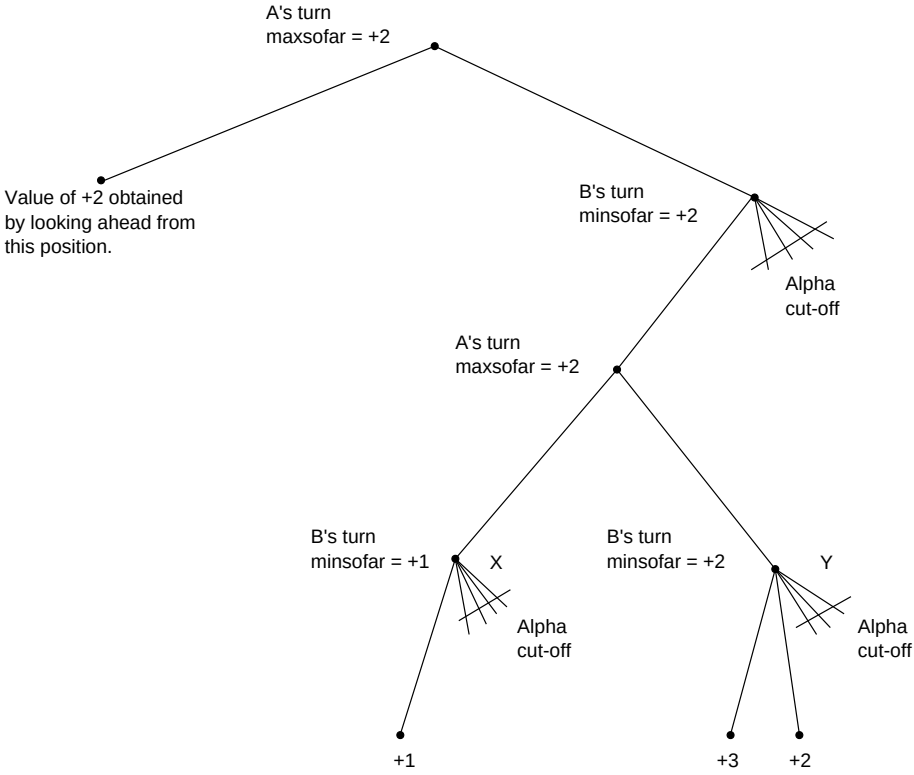
If a move at a B-position results in player B having another turn, the current alpha value can simply be passed on to the new position.

Similarly, in PROCtestAmove we require

```
3340    IF turn$="B" THEN v=FNminval(depth+1, maxsofar)
          ELSE v=FNmaxval(depth+1, beta)
```

Full scale alpha-beta pruning

As we have already remarked, alpha-beta pruning is rather more general than was indicated in the last section. Cut-offs can in fact take place as result of comparing 'minsofar' and 'maxsofar' values that are a long way apart in the lookahead tree. For example, part of a 4-leve1 lookahead tree is illustrated:



We assume that the left-hand branch of this tree has already been explored and that, as a rossult, the position at the top of the tree has a 'maxsofar' value of +2. If the program now explores the right-hand branch of the tree, it will encounter position X. The first move tried at position X gives X a 'minsofar' value of +1. We do not yet have a 'maxsofar' value for the position immediately above X, but the position at the root of the tree, three moves above X, does have a 'maxsofar' value of +2. Because the 'minsofar' value at X can only decrease, the final value of X cannot possibly affect the 'maxsofar' value (or the move chosen) at the root of the tree. The 'maxsofar' value from a position three moves above X can be used as an alpha value for pruning at position X.

In general, the alpha value for a position is defined as the maximum of all the 'maxsofar' values above that position in the lookahead tree. Similarly, the beta value of a position is the minimum of all the 'minsofar' values above that position.

To implement full scale alpha-beta pruning, each of our minimax functions must now have both an alpha and a beta parameter. For example, FNminval needs an alpha value so that it can test for alpha cut-offs. It also needs a beta value that can be compared with its own 'minsofar' value before an up-to-date beta value is passed on down the tree. FNmaxval needs both parameters for similar reasons.

The final versions of the relevant functions and procedures are:

```
2200  DEF PROCtryAmove
2210  LOCAL stones, v, capture, turn$
2220    stones = pit(p)
2230    PROCmakeAmove(p)
2240    IF turn$="B" THEN v=FNminval(1, maxsofar, 100)
          ELSE v=FNmaxval(1, maxsofar, 100)
2250    IF v>maxsofar THEN maxsofar=v : bestmove=p
2260    PROCmoveback(p, stones, capture, 0)
2270  ENDPROC
3000  DEF FNminval(depth, alpha, beta)
      : as  before
      :
3100  DEF PROCtestBmove
3110  LOCAL stones, v, capture, turn$
3120    stones= pit(p)
3130    PROCmakeBmove(p)
3140    IF turn$="A" THENv=FNmaxval(depth+1, alpha, beta)
          ELSE v=FNminval(depth+1, alpha, beta)
3150    IF v<minsofar THEN minsofar=v
3155    IF minsofar<beta THENbeta=minsofar
3160    PROCmoveback(p, stones,capture,7)
3170  ENDPROC
```

```
 3200  DEF FNmaxval(depth, alpha, beta)
       : as  before
       :

 3300  DEF PROCtestAmove
 3310  LOCAL stones, v, capture, turn$
 3320    stones = pit(p)
 3330    PROCmakeAmove(p)
 3340    IF turn$="B" THEN v=FNminval(depth+1,alpha,beta)
           ELSE v=FNmaxval(depth+1,alpha,beta)
 3350    IF v>maxsofar THEN maxsofar=v
 3355    IF maxsofar>alpha THEN alpha=maxsofar
 3360    PROCmoveback(p,stones,capture,0)
 3370  ENDPROC
```

Exercises

1 Use the timing version of PROCplayerA, suggested earlier, to test the effectiveness of the alpha-beta pruning techniques that have been described.

2 Alter the minimaxing functions defined for 'Last One Wins - or does he?' so that they use alpha-beta pruning. Arrange for the number of positions exaniined during a minimax search to be counted by the program. To do this, remember that each position is examined by a call of FNminval or FNmaxval. We therefore need the statement

```
count = count+1
```

at the start of each function. 'count' (not a LOCAL variable) is set to zero before the start of the search. Compare the number of positions examined by the program during a minimax search from various starting positions

(a) without alpha-beta pruning,

(b) with alpha-beta pruning,

(c) with alpha-beta pruning, but with moves being considered in the reverse order, ie. in the order 3,2,1, at each position.

Increasing the effectiveness of alpha-beta pruning

The effectiveness of alpha-beta pruning is highly dependent on the order in which moves are considered in each position in the lookahead. For example, if we use alpha-beta pruning in exploring the 'Last One Wins - or does he?' tree, the amount of pruning that takes place is considerably greater if we reverse the order in which the program considers the moves available.

If we look back at the diagram used to introduce alpha- beta pruning, we can see that pruning took place at position Q because the program had already tried a much better wve to position P. If position P had had a much lower value, then 'maxsofar' at the root of the tree would have been lower and no pruning could have taken place at Q. Alpha- pruning can be further enhanced by looking first at a good move for player B in the position at which pruning is going to take place. For example, pruning would have taken place earlier at position R if the second move had been considered first.

In the next diagram, pruning took place at position V because the best move for player Bat position T had already been tried. (Remember that high values are good for A, low values are good for B.) Thus pruning can take place only if a good move has already been tried at a position higher up the tree.

When fully effective, alpha-beta pruning reduces the branching factor of the lookahead tree explored to about the square root of its original value. For example, the number of positions in a Chess lookahead tree of depth 4 could, in theory, be reduced from over 800 000 to under 2000.

The only way that such spectacular pruning could be achieved would be if, at each position in the lookahead tree, the first move tried by the program were in fact the move that turned out to be best for the player whose turn it was at that position. Unfortunately, this information is not available until after the lookahead has been carried out!

Because of the large potential savings from alpha-beta pruning, it is important that, when evaluating a position, some attempt should be made to order the moves from the point of view of the player whose turn it is in that position. Any extra work involved in doing this will hopefully be offset by extra pruning.

One possibility would be to decide that capture moves tend to be good moves for the player whose turn it is. We could replace the single loop in each of FNbestmove, FNminval and FNmaxval, by two consecutive loops. In the first loop the program would try only the capture moves, and in the second it would try the other moves. This involves extra work in recognising the capture moves, but the extra work could be more than justified by enhanced alpha-beta pruning.

Many game playing programs order the moves in a position by carrying out a preliminary lookahead from <u>each</u> position encountered in the main lookahead tree. This preliminary lookahead might typically be of depth one or two. The moves available at a position are ordered on the basis of this preliminary lookahead and this ordering is used in considering moves for the main lookahead. It would be sensible if the work done generating new board positions during a preliminary lookahead was not repeated in the main

lookahead. To avoid this, it would be necessary to copy and store all the positions generated during a preliminary lookahead so that they could be examined later in the main lookahead.

The process of ordering moves at each position before continuing with a lookahead is often referred to as 'plausibility analysis'. The program attempts, usually on fairly superficial evidence, to decide which of the moves appear most plausible.

Other tree pruning techniques

In a game like chess with an average branching factor of 30, even with effective alpha-beta pruning, exploring a lookahead tree of any reasonable depth is impossible without further pruning. In the last section, we introduced the idea of a plausibility analysis for ordering the moves in a position before continuing the lookahead. This was done with the aim of increasing the amount of alpha-beta pruning that takes place.

Once a plausibility analysis has been done at a position, the results of the analysis can be used to implement further pruning. The program could be made to examine only the 'n' most plausible moves when continuing the main lookahead from that position. The value chosen for 'a' will determine how drastic the pruning is. One possibility is to give 'n' a fixed value at the start of the program. A more common arrangement is for the value of 'a' to decrease as the depth of the lookahead increases.

A plausibility analysis is usually fairly superficial. It must be stressed that pruning on the basis of such superficial evidence carries the risk that the program will eliminate a move from consideration when in fact a more detailed analysis would have shown that move to be a good me.

There is no such risk attached to alpha-beta pruning. The result of a lookahead with alpha-beta pruning will be exactly the same as if alpha-beta pruning had not been used.

Exercises

1 It was suggested earlier that improved alpha-beta pruning might be obtained if capture moves were considered first at each position in the lookahead. A capture can be recognised by making the move and examining the value of the variable 'capture', but this involves moving the pieces back if that move is not to be explored immediately. You could alternatively try to define an arithmetic test for recognising a capture move without actually making it. Implement and evaluate this modification. Two factors will be of interest: the number of positions examined in deciding on a move and the time taken to decide on a move. It will be interesting to see if fewer positions are examined than with the previous

version of the program. However, this increased pruning will be of only theoretical interest if the time saved by increased pruning does not compensate for the extra time taken to recognise capture moves.

2 If you succeeded earlier in defining an improved static evaluation function, then incorporate this in the final program. (If the function is too complicated, you may find that the program is slowed down by an unacceptable amount.)

3 Alter the program so that it keeps a complete record of the moves made during the course of a game and gives the user the option of seeing the game played through again after it is over. You will need to number the moves and use two parallel arrays to record whose turn it was at each move alongside the number of the pit from which the move was made.