12

Kalah on Atlas

A. G. Bell
Atlas Computer Laboratory
Chilton, Berkshire

I INTRODUCTION

This is a report on work done with the S.R.C. Atlas Computer at Chilton. The original intention was to demonstrate the on-line typewriter to visitors *via* a simple system which reacted to the user, in this case by refusing to be beaten twice in the same way at the game of Kalah.

The mechanism to achieve this is a memory, built up from information obtained in previous games, which is stored on magnetic tape. The program was designed to keep the size of this memory to small proportions by implementing two mechanisms the author believes to be commonly used by humans when solving problems. The two mechanisms are:

1. ignoring irrelevant information in the sense that, although it exists, it is highly probable that its precise structure or properties cannot alter the relevant information or *characteristics* of the problem being considered, and
2. accepting positions close to a solution or win, providing the opponent is further from a win.

Some of the difficulties of testing these ideas in practice are discussed and suggestions are made on how to overcome them, in particular with the game of solo whist.

II KALAH – THE RULES OF THE GAME

Kalah is an extremely ancient game, said to have originated in the Middle East. All that is required to play are 14 holes scooped in sand and the requisite number of pebbles. These pebbles need no distinguishing marks because it is the position of a pebble, or counter, rather than any intrinsic property, which denotes its possible move and ownership.

The more modern board and version of the game is given in figure 1. Each player controls a row of round *pits* on his side and the capsule-shaped bowl at his right called his *kalah*. The object of the game is to get the larger number of counters (playing pieces) in one's own *kalah*.

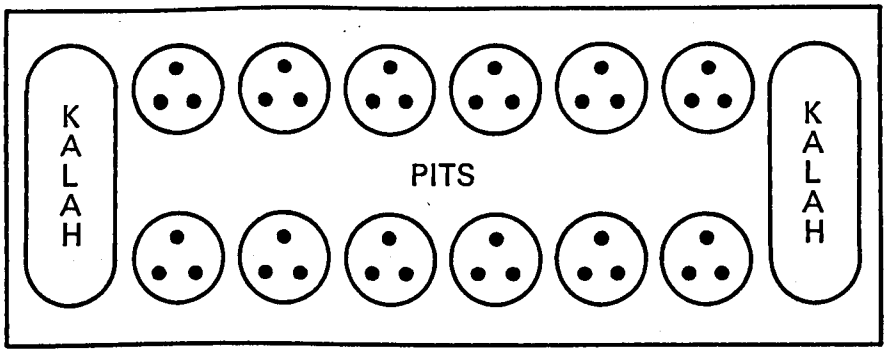

Figure 1

The number of counters used depends upon the time available and age of the players. For a short game and youthful players three counters are placed in each *pit* as shown. Six in a *pit* is generally agreed to make the most interesting game.

Players alternate in moving. Each player empties any one of his *pits* deemed advantageous, and, leaving it empty, distributes one counter into each *pit*, moving in an anti-clockwise direction. If there are enough counters to reach beyond his own *kalah* they are distributed one by one into the *pits* on the opposite side and then belong to the other player. The only place ever skipped is the opponent's *kalah*. Once in a *kalah*, the counters remain there until the end of the game.

The method of play, by distributing stepwise to the right, is subject to two simple rules:

1. If the last counter lands in your own *kalah*, you have another turn. By planning to have the right number of counters in two or more *pits* it is possible to have several turns in succession.
2. If the last counter lands in an empty *pit* on your own side opposite a non-empty opponent's *pit*, you capture all the counters in that *pit*, and place them, together with the one making the capture, in your own *kalah*. A capture ends your turn.

In reaching an empty *pit* on your own side, it makes no difference whether you have moved a single counter one space, or distributed all around the board and back to your own side. Indeed, if a *pit* contains 13 counters it is guaranteed to capture because the last piece goes back into the *pit* just vacated and the opposite *pit* must contain at least one piece.

The game ends when all the *pits* on one side are empty. The first player out usually loses because he receives none of the counters left in *pits* on the other side. They go into the *kalah* on the side of the player who has been able to save them. A good player will force the other to distribute and play out. On

the other hand, it is difficult to hold on to a lot of counters against a skilful player who tries to force their distribution.

III ILLUSTRATIVE GAME

This game was actually played against the program. The program is playing the top row and the author the bottom row (*see* figure 2).

The notation of the moves is as follows. The pits on the machine's side of the board are numbered M1 to M6 and on the player's side P1 to P6 (*see* figure 2.1). The pit chosen to be emptied is italicized and identified in the right hand column, and when a capture occurs the pit emptied is denoted in brackets.

A brief commentary on the game follows:

1. The author's opening move is the one claimed by Russell (1964) to be the best.
2. The program has 10 possible replies but has already rejected one from experience; this is its second attempt.
3. Unfortunately the author does not know how to stay in a winning position because this move, analysed later, is a loser.
4. The program takes full advantage of the position, i.e. capturing by emptying M6 in 4.5, otherwise it will lose.
5. The author threatens to capture M5.
6. The program therefore empties it and in return threatens to capture P5 for a crushing win. It has also calculated that it has a good chance to win or draw from this position irrespective of opponents play.
7. Pathetic.
8. The program continues on its winning course.
9. Resigns.

IV STRUCTURE OF THE PROGRAM

The program has, basically, 4 mechanisms. They are:

1. List legal moves.
2. Mini-maxing with simple evaluation.
3. Memory plus back-tracking to speed up accumulation of information.
4. Compression mechanisms to curtail size of accumulated information in memory.

IV.1 List legal moves

Because of the simplicity of the rules, the routine produced to calculate all the possible legal moves from a given position comprises about 100 Atlas Basic Instructions and runs extremely fast. This is an important property of the program because this routine is used a great deal in the look-ahead and back-track analysis, especially in the opening moves where it is unlikely to recognize any of the positions.

The program does not print its own continuation positions because they

		M1	M2	M3	M4	M5	M6		
1		3	3	3	3	3	3		
	MK 0							0 PK	P2
		3	3	3	3	*3*	3		
		P6	P5	P4	P3	P2	P1		
2		*3*	3	3	3	3	4		
	0							1	M1
		3	3	3	3	0	4		
3		0	3	3	3	3	4		
	1							1	P1?
		4	4	3	3	0	*4*		
4.1		0	3	*3*	4	4	5		
	1							2	M3
		4	4	3	3	0	0		
		continuation							
4.2		*1*	4	0	4	4	5		
	2							2	M1
		4	4	3	3	0	0		
		continuation							
4.3		0	4	0	*4*	4	5		
	3							2	M4
		4	4	3	3	0	0		
		continuation							
4.4		*1*	5	1	0	4	5		M1
	4							2	
		4	4	3	3	0	0		
		continuation							
4.5		0	5	1	0	4	*5*		
	5							2	M6
		(4)	4	3	3	0	0		c(P6)
		0	6	2	1	5	0		
5.1	10							2	P3
		0	4	3	*3*	0	0		
		continuation							
		0	6	2	1	5	0		
5.2	10							3	P1
		0	4	3	0	1	*1*		
		continuation							
5.3		0	6	2	1	5	0		
	10							4	P4
		0	4	*3*	0	1	0		

6.1		0	6	2	1	*5*	0		
	10							4	M5
		0	4	0	1	2	1		
		continuation							
6.2		*1*	7	3	2	0	0		M1
	11							4	
		0	4	0	1	2	1		
		continuation							
6.3		0	*7*	3	2	0	0		M2
	12							4	(claims
		0	4	0	1	2	1		win or
									draw)
7.1		1	0	3	2	0	0		
	13							4	P1
		1	5	1	2	3	*1*		
		continuation							
7.2		1	0	3	2	0	0		
	13							5	P5
		1	*5*	1	2	3	0		
		continuation							
		1	0	3	2	0	0		
7.3	13							6	P6
		1	0	2	3	4	1		
		1	0	3	2	0	0		
8.1	13							6	M1
		0	1	2	3	4	1		
		continuation							
		0	0	*3*	2	0	0		
8.2	14							6	M3
		0	1	2	3	4	1		
		continuation							
		1	1	0	2	0	0		
8.3	15							6	M1
		0	1	2	3	4	1		
		continuation							
		0	*1*	0	2	0	0		
8.4	16							6	M2
		0	1	2	3	4	1		
		1	0	0	2	0	0		
9	16							6	resigns
		0	1	2	3	4	1		

Figure 2

cannot be evaluated. When the last piece does land in its own *kalah* it buffers that position and starts from scratch with the next pit. Then, having investigated all the initial moves from the six pits, it recalls the continuation positions and either exhausts them or produces more continuation moves which go back into the buffer which is, in effect, a first in–last out stack. The result of this is that positions which are reached through many continuation moves will be the first to be investigated by the mini-max routine, and the cut off facility, in conjunction with the simple evaluation of position, (described in Section IV.2) should be enhanced.

IV.2 Mini-maxing with simple evaluation

Mini-maxing is a standard tool for playing full information games on machines. A good description of this tool is given by Michie (1966). In brief, the *kalah* routine looks $1\frac{1}{2}$ moves (or 3 plys) ahead by generating all the positions that can exist in that number of moves and ear-marks those positions it calculates to be to its own best advantage. However, as the opponent is assumed to evaluate positions in the same way and also has at least one intervening move, some of the positions are discarded as unattainable.

In order to prevent the evaluation of every position produced a simple cut off mechanism is incorporated. Consider figure 3. The routine takes the highest

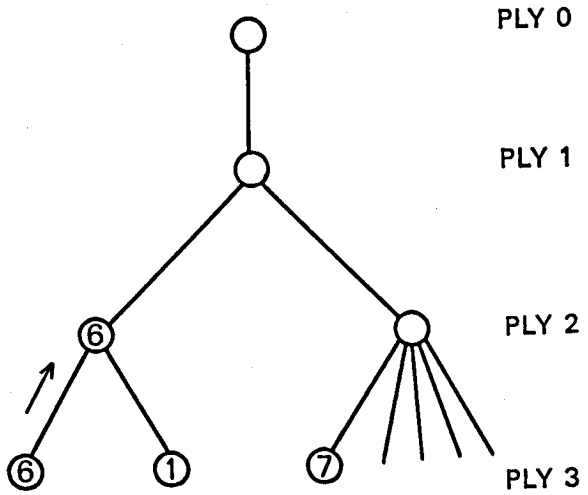

Figure 3. Positions are evaluated at PLY 3

value of the groups (in level 3) which are connected to the nodes in the second row, i.e. it maximizes. The highest of the left hand group of (6, 1) is 6 and this has been entered *via* the arrow. On investigation of the second group the routine finds that the first value it obtains (i.e. 7) is greater than the lowest value on level 2 (i.e. 6), and therefore can ignore all the remaining positions.

This mechanism works at any level, but the test of greater than or less than depends on whether the routine is at a maximizing level or a minimizing level.

The simple function chosen to evaluate positions at the 3rd level is to maximize $(Mk)-(Pk)$, i.e. to get more counters in the machine's *kalah* than the opponent can get in his own or, failing that, to keep the difference as low as possible. Actually this is a rather bad motivation because good players utilize a waiting game and the memory will have to over-ride some of the moves chosen eventually. Another weak feature of the program is that it will always accept the first victory it calculates or recalls. Therefore, the first move which will guarantee to obtain 19 counters in its own *kalah* is the one it will take and, indeed, is quite reasonable. Unfortunately, as its memory builds up, it tends to assess moves much earlier in the game and, instead of a brilliant, immediately crushing move, it will take the longest, dreariest path during which, as will be seen, the opponent has the chance to destroy the machine's illusion of being in an absolute winning position.

IV.3 Memory plus back track analysis

The information stored in the memory is a description of positions encountered which have a definite outcome, i.e. they are winning or losing positions. Before the first game the memory is empty and the moves chosen come entirely from the simple evaluation function with mini-maxing. Eventually the program realizes that it will either win or lose in the next $1\frac{1}{2}$ moves. If it wins it only records its present position and tags it as a winner, i.e. a good position to try for, if given the chance, in a later game. If, however, it loses, it attempts

(*a*) to find out why, and

(*b*) to build up its store of winning/losing positions.

Both these aims are attempted by back-track analysis. This is simply a short investigation of the region of the game tree in which the program finds itself. The argument is that, now it has eliminated a previously chosen move because it inevitably lost, it can consider more deeply the alternatives to that move and possibly even find a win. Also, by looking at positions closely related to its losing position, more winning/losing positions should be produced which are worth storing. By recording the course of the game the program can return to the position one move before it inevitably lost and can assess that losing position from its memory. Hence it will choose differently. A count is set to evaluate 100 nodes in that region of the game tree, and, whenever all the nodes have been evaluated at the lower level, the previous position played in the actual game is recalled and the back-track analysis continues. This results in the program, having terminated this activity, never playing the same game twice and also being unpredictable at what point in the game it will diverge from its previous play. This rote learning with associated back-up scores is well described by Samuel (1960). Figure 4 shows clearly that once the node $\otimes$ has been evaluated and the same board position is found in a later game, then its score has, in effect, already been backed up by 3 levels, and if it

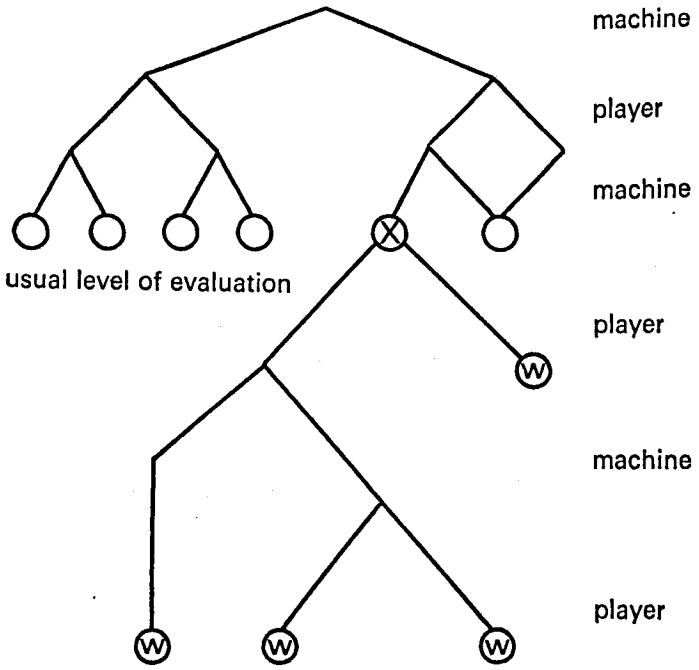

Figure 4. W is a winning position, hence x is recorded in the memory as a winner and the machine will prevent the opponent having it

becomes effective in determining the move to be made, it is a 6 ply (i.e. 3 move) score rather than a simple 3 ply (i.e. $1\frac{1}{2}$ move) score.

The method by which positions in the memory are matched against positions in actual play is by a simple table look up. The initial method chosen was by hashing (*see* Hopgood, 1966) or forming a computed entry on the contents of the two *kalahs*. This method was discarded because the precise contents of the opponent's *kalah* is, quite often, irrelevant as will be shown later. The simpler table look up method was therefore used. To understand the layout of the memory requires an explanation of the author's original aim. This was that if the program was given the opening move in 3-in-a-pit *kalah*, it would be unbeatable, i.e. knowing that (according to Russell) position 1 in the illustrative game (figure 2) is the winning opening, all the program has to learn is why each of the ten possible replies fails. The point is that the memory is designed to record positions of 3-in-a-pit *kalah* and although it will play the higher order games, it does not record them but plays using only the evaluation function.

When a winning or losing position is discovered the program compacts it, marking it as a winner or loser, and then multiplies the contents of its own

kalah by 2^6. This then points into a table of 1280 entries (a cut off is made if MK>20). It still entails a linear scan to be made but the entry could be improved by using the contents of the first one or two pits next to the *kalah* and hashing them.

Positions are stored, then, in the manner described and when the program is considering other positions it similarly compacts them, calculates the entry point into the table, and compares the position with those in the memory which have the same *kalah* contents, ensuring that all the pits' contents also agree. If a position is recognized, then the program will either accept it if it is a winner or reject it if it is a loser. The memory is split into ten parts each with 1280 entries and the program checks which of the ten replies the player has chosen and, in order to confine the game, reads that part of the total memory down. This, incidentally, is not a good thing to do, as the author attempts to explain later, nor is it wise to assume that 1280 entries is sufficient to hold the required information. These features are described because, once implemented and the program discovered that one of the ten replies was a loser, the aim of the project developed from playing an unbeatable game of *kalah* to studying ideas and techniques which may be applicable to a range of problems.

IV.4 Memory compression mechanisms

In the introduction, two memory compression mechanisms are mentioned. It is worth adding that the simple evaluation of position function mentioned in section IV.2 also restricts the size of the memory in the sense that there are positions from which the machine could choose many moves which still guarantee a win. The evaluation function tends to take the shortest route and hence many plodding paths and positions to a win are ignored. However, the two mechanisms considered by the author to be more akin to the way humans treat games of this type are called *characteristics* and *parameters*.

Characteristics. Consider figure 5.1. The machine is playing the top row and is presented with this position which is a winner (figure 5.2) *via* the moves M1 (continue) M2 (continue) M1. The machine will therefore store this position as a winner but it does not need to store the contents of every pit because it is immaterial to the winning sequence what the other pits contain (they may contain any legal permutation of the remaining counters in the pits which have not changed their contents). Similarly in figure 5.4 the winning sequence of moves is M6–M1–M3–M1–M2–M1–M4 (capture). The quick way to extract the required information is by non-equivalencing the compressed position before the winning move with the compressed position after the winning move. Non-equivalencing two equal values gives the answer zero; two dissimilar values give a non-zero answer. Thus, non-equivalencing figures 5.1 and 5.2 gives the answer in figure 5.3 of which pits are relevant or *characteristic* of the win. These are M1 and M2 plus the *kalah* (which must have a minimum value of 16).

Unfortunately this is not sufficient in every case, for if this operation is

5.1	16	1	2	0	0	0	0	16
		1	0	0	0	0	0	
5.2	19	0	0	0	0	0	0	16
		1	0	0	0	0	0	
5.3	16	1	2	x	x	x	x	x
		x	x	x	x	x	x	
5.4	7	0	0	2	0	0	6	16
		0	0	5	0	0	0	
5.5	19	0	0	0	0	1	0	16
		0	0	0	0	0	0	
5.6	7	x	x	2	x	0	6	x
		x	x	5	x	x	x	
5.7	7	0	0	2	0	0	6	x
		x	x	5	x	x	x	

Figure 5

carried out on figures 5.4 and 5.5 then the result shown in figure 5.6 is incorrect, i.e. 6 counters in M6, 0 in M5, 2 in M3, a minimum value of 7 in the *kalah* and at least 5 counters in the pit opposite M3. The following additional statement must therefore be included: 'Any pit to the right of an altered pit on my side may also be relevant to the winning position and consequently must be stored'. This results in the position given in figure 5.7.

The extraction of relevant features always guarantees that the contents of at least 5 of the opponent's pits plus his *kalah* will be ignored plus the fact that, at the most, two pits can be assigned minimum values for their contents.

The author emphasizes that this extraction mechanism is heavily dependent on the features of the game of *kalah*. In a more general framework one should consider removing pieces from the board until it can be shown that their presence is essential to the win.

Parameters. The second mechanism, called *parameters*, is a misnomer, but the principle involved is simple. It is akin to the queen being captured in chess for no loss, i.e. one of the players gets so far ahead in the game that the opponent will retire before reaching an actual losing position.

The program records all the previous positions in a game. When it wins or loses, it scans the contents of each *kalah* in all the previous moves, and places

M_k	P_k
19	18
18	18
17	9
16	4
15	2
14	2
13	1
12	1
11	0
10	0

Figure 6. Relative values of *kalahs* for pseudo winning-losing positions (rounded to nearest integer)

a 90 per cent bias on the results. An example (using an actual value achieved by the program) is that if the program can obtain 17 counters in its *kalah* and its opponent has only 9, then 90 per cent of the time (or better) the game has been won from that *kalah's* relation. We define the relation to be nearly as good as a win, and it will therefore be stored in the memory. The actual relative values produced in the machine are given in figure 6. Due to an error in the program, it classes draws to be as good as wins, but to correct this delusion would entail altering the program and destroying the memory already accumulated. This mechanism does not conflict with the *characteristics* mechanism and it reduces the size of the tree to be searched. We again note that the content of the opponent's *kalah* is still not relevant information to be stored in the memory. The position given in figure 7 has good prospects providing the

	1	2	0	0	1	0	
13							4
	6	4	2	0	2	1	

Figure 7

opponent has 4 or less counters in his own *kalah*; otherwise it is assessed by the evaluation function. In the example given, the program would accept the position at either of its maximizing levels as equivalent to a win despite the fact that the opponent has an intervening move (even if the position is reached) before the certain end of the game. In fact, the example is 'cooked' and the opponent has a crushing set of 15 continuation moves to a certain win. This will modify the program's opinion that 16 against 4 is a successful ratio.

It is now obvious that the decision to split the memory into ten parts is a bad thing, for the program has already stored positions which will be useful when it comes to play the other replies. One solution to this problem is to tell the program the correct moves for the opening game, to let the evaluation function play the middle game, and to rely on the *characteristics/parameters* memory to play the end game.

V RESULTS

In reply to the best opening move for 3-in-a-pit *kalah* (figure 2.1), the program initially chose the position shown in figure 8, i.e. M3–M2. The total number

	5	0	0	3	3	4	
2							1
	4	4	3	3	0	4	

Figure 8

of possible positions which can be reached from this configuration is over 2000. With the feeble goal of the evaluation function, many of the paths can be ignored because the program tends to stay on the path to the quickest win.

The memory generated without *characteristics* or *parameters* numbered just over 400 positions in order to prove that this reply will inevitably lose. To build up this memory quickly entailed the author playing the program, beating it, checking how much of the game had been analysed by the back-track analysis, and then playing the program through moves it had still not analysed. Thus the program learnt quickly and this procedure could be used off-line.

When the *characteristics* mechanism was added the memory was only reduced to about 300 entries. This seems disappointing until it is remembered that the positions being recorded will be useful when replies to other opening moves are studied – that is, the program is, to a certain extent, analysing the end game of *kalah*.

Finally, when the *parameters* have been calculated, the memory drops to about 150 entries but the program becomes more likely to claim a win, and then to lose. Surprisingly, people seem to enjoy this aspect of the program because if it thinks it will win, it says so, and, if a good waiting game is then played, the program can lose badly.

Having found that the position in figure 8 is a losing response the program next chose the position in figure 2.2 (the illustrative game) and trouble was encountered because the author did not know the correct response to this move. Consequently the memory began to build up to the danger point of the program believing that 9 times out of 10 it will win from this position which, of course, affects the way it plays that part of the game it has already solved. The unfortunate state of affairs has now been reached where the program, having learnt to play against a good opponent, now begins to deteriorate against a bad opponent. In fact, in demonstrations to visitors, the memory built up is not retained because the program can and does win from appallingly bad positions which may change the *parameters* drastically.

VI OTHER GAMES

The work described in this paper would seem to be applicable to other games, in particular that of solo whist. This card game is suitable because, quite often, many of the cards held by each of the four players are relatively unimportant and it would appear that the compression mechanisms may be effective in this game (which differs markedly from *kalah* in that it is not a full information game). Another advantage of teaching a program to call and play solo whist is that it is easy to write a program to defend calls or contracts by giving full information to the program which defends the contract, i.e. the program knows where the voids are, knows the boss cards and if a finesse is worth while, etc. The trick is to have the program which has to learn the game play against this simple but powerful game teacher. This should save a great deal of time involved in humans having to play the program, or inspect its memory, in order to teach it what configurations of cards are worth calling contracts on and in what order to play the cards. This project has been started.

REFERENCES

Hopgood, F.R.A. (1966), Hash tables. *A.C.T.P. summer school notes.*
Michie, D. (1966), Game-playing and game-learning automata. *Advances in programming and non-numerical computation* (ed. Fox, L.) London: Pergamon Press.
Russell, R. (1964) Kalah–the game and program. *Stanford artificial intelligence project, Memo. No. 22.*
Samuel, A.L. (1960), Programming computers to play games. *Advances in computers 1.* New York: Academic Press.

o