SOLVING KALAH

Geoffrey Irving[1] *Jeroen Donkers and Jos Uiterwijk*[2]

Pasadena, California Maastricht, The Netherlands

ABSTRACT

Using full-game databases and optimized tree-search algorithms, the game of Kalah is solved for several starting configurations up to 6 holes and 5 counters per hole. The main search algorithm used was iterative-deepening MTD(f). Major search enhancements were move ordering, transposition tables, futility pruning, enhanced transposition cut-off, and endgame databases.

1. INTRODUCTION

Kalah is a modern, commercial variant of Mancala, introduced in the 1950s by a firm called "The Kalah Game Company" (owned by W.J. Champion). It has gained a large popularity especially in the United States and is still played in pubs and at home. In 1960, a first computerized version of the game was produced and many others followed. Remarkably, Kalah has a relatively long history in Artificial Intelligence: Bell (1968) already used Kalah to demonstrate game playing by a computer, and Slagle and Dixon (1970) used Kalah to illustrate their M & N search algorithm. Nowadays Kalah is often used as an example game in computer-science courses.

The term 'mancala' is used to indicate a large group of related games that are played almost all over the world (Murray, 1952; Russ, 2000). Mancala games (also known as 'pebble-and-pit games' or 'count-and-capture games') are played on a board that contains 2, 3 or 4 rows of holes. Sometimes these holes are simply dug in the soil or drawn on paper. Often there are two or four additional holes (called *stores*) with a special meaning. The games are usually played by two players, although one-player and three-player versions are known. Mancala games are played with a large set of equal counters. These counters can be pebbles, shells, seeds or any small round objects. The game starts with a certain distribution of the counters over the pits (usually an equal number per hole). A move is made by selecting one of the holes, lifting all counters out of it and putting back the counters one-by-one in adjacent holes in a certain direction. This is called 'sowing'. The hole in which the last counter is put determines what happens next. Sometimes a capture takes place and the turn is over, sometimes the sowing continues, and other times the player is allowed to do another move. The goal of the game is always to capture as many counters as possible.

There is a variety of board sizes for mancala games and there are even more variations in the rules. Some of the games are very easy to play while others are extremely difficult to master. Murray (1952) and Russ (2000) group the mancala games together by the number of rows on the board and also by some specific rules. Especially important is the capture rule. In one group of mancala games a capture is allowed if the last counter is put in an opponent's hole that contains 1 or 2 counters. These are mostly African games. The well-known game of Awari (played on the Computer Olympiads) belongs to this group. In another group of games, a capture is allowed if the last counter is put in an empty hole on the player's side. This group is mainly played in South East Asia, but Kalah belongs to this group too.

The goal of this paper is twofold. First, we describe how the game of Kalah was solved *strongly* (according to Allis' (1994) definition) for small instances using full-game databases, and *weakly* for larger instances using game-tree search. Second, we present the results of a more theoretical study of the game of Kalah.

The rest of this paper is organized as follows. Section 2 concentrates on the complexity of Kalah and on

[1]Student at Computer Science Dept., California Institute of Technology, Pasadena, CA 91125, USA. E-mail:irving@caltech.edu.
[2]Dept. of Computer Science, Universiteit Maastricht, Maastricht, The Netherlands. E-mail:{uiterwijk,donkers}@cs.unimaas.nl.

properties of Kalah game graphs. Section 3 describes the techniques used to solve Kalah. Section 4 gives the results, and Section 5 contains a discussion and some indications of future research.

2. ANALYSIS OF KALAH

The game of Kalah has a number of properties that influence the way of how the game can be solved. The first property to mention is that a board position is unique in a game, i.e., there is no repetition of positions. This is not true for all mancala games. For instance, Awari can have move cycles. The second property is that captured counters in Kalah do not re-enter the game. This means that the game value of a certain position can be derived from the configuration of active counters only. (These are counters that are not captured.) If we express the game value as the total number of stones that a player can capture from a given position, then this value is equal to the number of stones already captured plus the stones that can be captured from the active stones. It means that endgame databases only based on the configuration of active counters can be built. The game of Awari has this second property too; the construction of an endgame database for Awari is similar to that of Kalah endgame databases.

2.1 Kalah Rules

Kalah is played on a board with two rows of 6 holes and two stores, called *kalahah*. The two players (North and South) sit at each side of the board. We assume that South always starts the game. The game usually opens with 4 counters in every hole, but other amounts (2, 3, 5, 6) are possible. A move is made by selecting a non-empty hole at the player's side of the board. The counters are lifted from this hole and sown in anti-clockwise direction, starting with the next hole. The player's own kalahah is included in the sowing, but the opponent's kalahah is skipped. In some implementations of Kalah, the hole from which the sowing is started is skipped too (in the case that more than twelve stones are to be sown), but we will refrain from this rule.

There are three possible outcomes of a move: (1) if the last counter is put into the player's kalahah, the player is allowed to move again (such a move is called a *Kalah-move*); (2) if the last counter is put in an empty hole on the player's side of the board, a capture takes place: all stones in the opposite opponent's pit and the last stone of the sowing are put into the player's store and the turn is over; and (3) if the last counter is put anywhere else, the turn is over directly. The game ends whenever a move leaves no counters on one player's side, in which case the other player captures all remaining counters. The player who collects the most counters is the winner.

In our investigations we also considered Kalah with less than 6 holes and with different initial numbers of counters. Therefore, we will use the notation Kalah(m,n) to indicate Kalah with m holes per side and initially n counters per hole.

2.2 Complexity

Although the rules of Kalah are simplified so that it is easier for humans to play Kalah than it is to play other mancala games, the complexity of Kalah in terms of computer game playing is larger than that of some comparable traditional games. For instance, the traditional game of Dakon is very similar to Kalah, but is far more difficult to play for humans. However, it appears that Dakon has winning openings that end the game in the first turn (Donkers, de Voogt, and Uiterwijk, 2000b). These openings are easily found by a computer, but were only recently found by hand.

The difference in complexity of mancala games like Kalah as perceived by humans and computers is mainly caused by the difficulty of counting the number of stones that are put in a pit. In most traditional scenarios, the players are not allowed to count the stones in a pit. If there are more than, say, ten stones, the player has to memorize the exact number of stones (traditional mancala games are imperfect information games). This is one of the factors that make the game attractive to play. For a computer, the difficulty of counting does not exist. Many computerized versions of mancala games actually help the human player by displaying the exact contents of the pits.

For a computer, the complexity of (solving) Kalah is lower than that of Awari. The board sizes of both games

and the number of stones are equal, but in Kalah the number of stones that can be captured in one move can be larger than in Awari. This means that Kalah takes fewer moves to be finished. Also the final phase of the game is easier in Kalah because in Awari a player must always leave the opponent a move if possible, but in Kalah a player is allowed to take all opponent's stones as soon as possible. Furthermore, Awari endgames can contain move cycles whereas these are impossible in Kalah. Below we provide an estimate of the numerical difference in complexity between Awari and Kalah.

Table 1 gives the average game length (d), average branching factor (w) and the estimated game-tree complexity (w^d) for Kalah(m, n). The estimates were obtained by playing 100 games per instance between two players that use plain $\alpha\beta$ with a 7-ply depth and that order equivalent moves randomly. The instances Kalah(1, n) are left out because the branching factor of these instances is 1. The results for Kalah(2,n) are not very meaningful, because both players were able to solve the game at the 7-ply depth. The state-space complexity of Kalah(m, n) is given in Table 2.

$m\backslash n$		1	2	3	4	5	6
2	d	3.00	8.00	6.00	12.00	6.00	13.00
	w	1.40	1.34	1.50	1.47	1.72	1.64
	w^d	2.74×10^{0}	1.04×10^{1}	1.14×10^{1}	1.02×10^{2}	2.59×10^{1}	6.21×10^{2}
3	d	6.42	9.00	13.35	13.58	14.00	15.05
	w	1.48	1.72	2.04	2.21	2.25	2.39
	w^d	1.24×10^{1}	1.32×10^{2}	1.36×10^{4}	4.75×10^{4}	8.52×10^{4}	4.95×10^{5}
4	d	7.00	10.00	18.25	23.05	22.28	26.72
	w	1.72	2.21	2.67	2.95	3.11	3.13
	w^d	4.45×10^{1}	2.78×10^{3}	6.08×10^{7}	6.75×10^{10}	9.52×10^{10}	1.74×10^{13}
5	d	10.04	15.00	23.34	30.04	35.25	37.73
	w	2.06	2.67	3.18	3.51	3.81	3.94
	w^d	1.42×10^{3}	2.50×10^{6}	5.33×10^{11}	2.40×10^{16}	3.00×10^{20}	2.94×10^{22}
6	d	11.49	16.76	26.62	30.75	37.66	49.25
	w	2.78	3.43	3.65	4.08	4.51	4.75
	w^d	1.26×10^{5}	9.37×10^{8}	9.29×10^{14}	6.00×10^{18}	4.33×10^{24}	2.12×10^{33}

Table 1: The estimated game-tree complexities (w^d) for Kalah(m, n).

$m\backslash n$	1	2	3	4	5	6
1	20	70	168	330	572	910
2	252	2,574	12,376	40,698	106,206	237,510
3	3,432	100,776	961,400	5,259,150	20,590,944	64,448,228
4	48,620	4,085,950	77,134,200	700,687,130	4.11×10^{9}	1.80×10^{10}
5	705,432	169,344,630	6.31×10^{9}	9.52×10^{10}	8.36×10^{11}	5.12×10^{12}
6	10,400,600	7.12×10^{10}	5.25×10^{11}	1.31×10^{13}	1.73×10^{14}	1.48×10^{15}

Table 2: The state-space complexity for Kalah(m, n).

Although Kalah(6, 4) has the same state-space size as Awari (about 10^{12}), the estimated game-tree complexity is much lower (6×10^{18}) than that of Awari (10^{35}) (Allis, 1994). This is mainly due to the game length. Awari has an average game length of 60, according to Allis (1994), and Kalah(6, 4) only of 30. The average branching factor of Awari is smaller (3.5) than that of Kalah(6, 4), which is more than 4. The estimated complexity of Kalah(6, 4) is most comparable to that of Connect-Four.

Kalah(6, 6) has almost the same game-tree complexity as Awari (2×10^{33}), but has a higher state-space complexity (10^{15}). The complexity of this instance of Kalah is most comparable to that of Checkers.

2.3 Full-game Databases

To investigate the complexity of Kalah, full-game databases were constructed for smaller instances of Kalah. These databases were constructed in two stages. In the first stage the complete game graph was built starting

from the opening position. In the second stage the game values of all positions in the graph were computed backwards. Table 3 gives the sizes of the game graphs and the amount of the state space that is occupied by the game graphs. The largest database that was built was that of Kalah(4, 3), which contains 4,604,996 entries. Up to this size, a major part of the computations could be done in internal memory. For larger instances, computing the full-game database needs extra disk access which slows down the computation considerably. The generated full-game databases are available on the computer-games page at `www.cs.unimaas.nl`.

$m\backslash n$	1	2	3	4	5	6
1	2	2	2	3	4	6
	10%	2.9%	1.2%	0.9%	0.7%	0.7%
2	8	24	138	168	58	2482
	3.1%	0.9%	1,1%	0.4%	0.1%	1.1%
3	73	2,941	24,936	190,579	755,748	1,255,339
	2.1%	2.9%	2.6%	3.6%	3.7%	2.0%
4	880	226,774	4,604,996			
	1.8%	5.6%	6.0%			
5	11,465					
	1.6%					
6	178,708					
	1.7%					

Table 3: Game-graph sizes and state-space occupation for Kalah(m, n).

Table 3 shows that a major part (say 94% or more) of the configuration space is not reachable in the game. However, it is very difficult to extrapolate these numbers to larger Kalah instances. A conservative estimate of the size of the game graph for Kalah(6, 4) would be about 1.3×10^{12}. The average branching factor of the nodes in some of the game graphs is given in Table 4. These numbers suggest that the estimations given in Table 1 are plausible.

m	n	branching factor	m	n	branching factor
3	1	1.18 $\pm$0.90	4	1	1.29 $\pm$0.96
3	2	1.36 $\pm$0.85	4	2	2.13 $\pm$0.95
3	3	1.72 $\pm$0.85	4	3	2.62 $\pm$0.89
3	4	2.02 $\pm$0.81	5	1	1.63 $\pm$0.96
3	5	2.16 $\pm$0.78	6	1	2.04 $\pm$1.03
3	6	2.23 $\pm$0.76			

Table 4: Branching factors ($\pm$ standard deviation) in game graphs of Kalah(m, n).

Table 5 shows the game values (Win/Loss/Draw) that were found for the smaller Kalah games above (a more complete overview is given in Table 10). Table 5 also gives the distribution (in percentages) over the game values (W/L/D) of all the nodes in the game graphs. Obviously, there is a correlation between the actual game value and the value of the majority of the nodes, but there are five instances where the actual game value differs: Kalah(2, 3), Kalah(3, 2), Kalah(4, 3), Kalah(5, 1), and Kalah(6, 1).

$m\backslash n$	1	2	3	4	5	6
1	D 0/0/100	L 0/100/0	W 100/0/0	L 0/100/0	W 100/0/0	D 0/0/100
2	W 38/38/24	L 33/46/21	L 42/32/26	L 16/61/23	W 76/7/17	W 45/36/18
3	D 23/37/40	W 38/44/18	W 46/40/14	W 46/42/12	W 46/43/11	L 35/45/10
4	W 37/35/28	W 44/44/12	W 44/46/10			
5	D 34/45/21					
6	W 38/44/18					

Table 5: Game values of Kalah(m, n) and the distribution (in percentages) of node values (W/L/D) in the game graphs.

2.4 Game Graph of Kalah(4, 3)

The game graph of Kalah(4, 3) is the largest one that we constructed. It is probably the best approximation of the game graphs for the larger instances of Kalah. Game graphs of Kalah can be partitioned into subgraphs on base of the kalahah contents. We define the subgraph $g(p, q)$ as the graph of all positions in which South has captured p counters and North has captured q. The subgraphs themselves are ordered in layers $l(r)$, each layer containing subgraphs $g(p, q)$ with the same amount of total captured counters ($r = p + q$). The layers are ordered from top to bottom, starting with the top layer $l(0)$ that only contains the subgraph $g(0, 0)$.

Vertices in the game graph exist only within a subgraph or from a node in a layer to a node in a layer *below*. No vertices exist between subgraphs of the same layer, nor to a higher layer.

In Table 6 counts for the first six layers of the game graph for Kalah(4, 3) are presented. For every subgraph, the table gives the distribution of the node game values. The first observation to be made is that the number of Win and Loss nodes in each subgraph is not very different, except for the subgraphs on the left and right end of each row. The second observation is that sometimes the majority of the nodes have a game value that is in contradiction with the distribution of the captured counters. For instance, in subgraph $g(5, 0)$, South has already captured 5 counters and North has captured none, but the game is a sure loss for South. Less extreme is the situation in subgraph $g(4, 2)$ (layer $l(6)$). Here, South captured 2 counters more than North, but 10,341 nodes are a loss for South and only 6,500 are a win. These 'anomalies' happen mostly in the layers $l(6)$ to $l(10)$ of the graph. The game graph suggests that the often-used heuristic of capture difference (material balance) might not work very well in Kalah.

	Captured by South						
Layer	0	1	2	3	4	5	6
$l(0)$	2/1/0						
$l(1)$	1/6/0	5/6/1					
$l(2)$	1/15/0	36/41/9	11/8/1				
$l(3)$	0/20/0	153/163/30	181/168/43	6/5/3			
$l(4)$	0/17/0	412/376/77	1514/1280/293	293/382/72	4/7/0		
$l(5)$	0/8/0	689/609/108	6183/3796/1057	3661/5299/1026	351/413/63	0/4/0	
$l(6)$	0/3/0	821/708/125	13446/8056/2116	22204/22149/4908	6500/10341/1730	274/292/39	0/1/0

Table 6: First six layers in the game graph of Kalah(4, 3). The rows indicate the layers, the columns indicate the number of counters captured by South. Per cell, the number of Win/Loss/Draw nodes (from South' perspective) is given.

3. SOLVING LARGER INSTANCES OF KALAH

Larger instances of Kalah(m, n) ($4 \leq m \leq 6$, $1 \leq n \leq 6$, except 6(6)) were solved with an optimized tree searching program (written in C++). The central algorithm is MTD(f) with iterative deepening. Optimizations include move ordering, transposition tables, futility pruning, and an endgame database.

The evaluation function was calculated by subtracting the numbers of counters in the two kalahahs. For simplicity and speed, this function was also used at all interior nodes as well as leaf nodes, though only the values at leaf nodes determine the full-depth game value.

MTD(f), or Memory-enhanced Test Driver, is an improvement on alpha-beta pruning using only zero-window windows in the search (Plaat *et al.*, 1996a). The iterative-deepening step size used was 3. However, in order to reach full depth faster, the algorithm uses interative deepening only to depth 30, then proceeds to a complete exhaustive search. At least for Kalah(6, 3), this is advantageous because most lines of play finish before this depth, and thus a search to depth 32 duplicates much of the work done in a depth-29 search. Strangely, even though the Kalah(6, 4) and Kalah(6, 5) games are longer than 30 ply, it is still highly inefficient to increase this cut-off depth. In fact, both Kalah(6, 4) and Kalah(6, 5) were solved faster without any iterative deepening, provided the first guess given to MTD(f) was the correct value. Therefore, it seems that the major advantage of iterative deepening here is not to provide move-ordering knowledge, but to provide a close guess to the true minimax value for MTD(f).

The move-ordering sort was based on four factors (in order of precedence): transposition-table suggestions, extra turns, captures, and right-to-left default ordering.

The transposition tables used employ full collision checking, so no mistakes are allowed. Moreover, the transposition table stores only the numbers of active counters, since the effects of the kalahahs on the final score are trivial. Theoretically, this allows multiple positions to use the same hash-table entry, though it is doubtful that this creates any significant tree reduction.

The hash function used together with the transposition table was written by Bob Jenkins (1997). The standard hash function for game playing is the Zobrist (1970) hash function, which has the advantage of incremental computation. In Kalah, this advantage disappears due to sowing that causes changes all over the board. The small board size allows quick computation using the Jenkins hash function. This hash function is especially built to hash varying-length strings, but it can take all kinds of input. The function takes $6n + 35$ instructions to hash a string of n bytes, which is cheaper than most other hash functions when $n \geq 10$. The Jenkins hash function produces 32-bits hash values. The core of the Jenkins hash function is the mixing of three 32-bits integers (a, b, and c). At the start of the computation of the hash value for a given string, c is initialized with the previous hash value (or an arbitrary value at the first call). Then a, b, and c are incremented with bytes 1-4, 5-6, and 7-12 of the input string, respectively. Now the three integers are mixed using subtraction, bit-XOR and bit-shift operations. This process is repeated with the rest of the input string. At the end, c is returned as the new hash value.

For Kalah, the mix procedure is called only once per position. Integers a and b represent the hash position, with 5 bits per active hole. Note that this restricts active holes to fewer than 32 counters, and the rare positions that violate this requirement are excluded from hashing. The initial values of a and b, which uniquely encode the active state, are stored in the hash table for full collision checking. Without full collision checking, it is possible that faulty information could be introduced into the algorithm.

The Kalah program also uses a version of futility pruning (Schaeffer, 1986). At each node, the possible range of scores at the end of the game is calculated by adding all stones in play to both player's kalahahs. This range is then improved slightly with knowledge of captures, etc. If the range does not intersect the alpha-beta window, an immediate, full-depth cut-off is possible. Note that unlike some implementations of futility pruning in chess, futility pruning in Kalah is completely safe. The effectiveness of futility pruning depends on whether endgame databases (see Section 3.1) are used, and is summarized in Table 7.

Databases?	3 stones	4 stones
yes	6%	15%
no	38%	

Table 7: Tree reduction due to futility pruning with and without endgame databases.

Enhanced transposition cut-off is a method of improving transposition-table use by first searching all successor nodes for transposition-table data (Plaat *et al.*, 1996b). Applying the same idea to futility pruning and endgame look-ups, and then using the resulting information to improve move ordering reduced tree size by a factor of up to 8, and cut total running time by about 3 when solving Kalah(6, 4).

One optimization that did not result in significant tree reduction was the history heuristic (Schaeffer, 1983, 1989). Here, the decrease in tree size was counteracted by the increased time required to search individual nodes. This is probably due to the simplicity of Kalah compared to chess, where the history heuristic is quite beneficial.

3.1 Endgame Databases

The endgame databases for Kalah were built using a form of retrograde analysis, analogous to the way in which endgame databases were (and still are being) built for Awari (Allis, van der Meulen, and van den Herik, 1991; Nievergelt *et al.*, 1995; Lincke, 2000; Lincke and Marzetta, 2000). Because the number of counters that can be captured from a certain position in Kalah only depends on the active counters, endgame databases for Kalah can be built totally on the base of the configuration of the active counters. Captured counters do not re-enter the game, so the value of a position with n active counters can be determined by checking positions

with less than n active counters. Starting with 2 active counters, separate databases were created for every number of active counters, up to 20 counters. In contrast to the approach by Allis *et al.* (1991), the values of the positions were not calculated by performing *reverse* moves, but by performing a *forward* minimax search for every position. It is costly in mancala games to compute reverse moves. The costs of the forward minimax searches are compensated by the ease of generating moves forwardly.

The set of configurations in Kalah with n active counters on a board with m holes per side can easily be ordered and enumerated using basic combinatorics. Of course, all positions that have no counters at the side of one of the players are left out. The size of this set is:

$$Size(m,n) = \binom{n+2m}{2m} - 2\binom{n+m}{m} + 1 = O(n^{2m})$$

Positions are indexed without regard to North and South by differentiating between the player to move and the waiting player. Using the combinatorial ordering of these configurations, endgame databases can be constructed that contain no unused space. However, these configurations includes numerous positions which are unreachable or unimportant. The advantage is that no look-up is required to determine whether a position is in the database. At present, the entire endgame is loaded into RAM. Only 4 bits are used per entry representing the score of the player to move, and a lower bound is stored if necessary. Table 8 gives an overview of the sizes of the endgame databases for Kalah(6, n).

Counters	Positions	Size (MB)	Counters	Positions	Size (MB)
17	51,694,042	24.6	24	1,250,490,151	596
18	86,224,034	41.1	25	1,851,010,435	883
19	140,766,326	67.1	26	2,705,662,765	1290
20	225,332,381	107	27	3,908,582,301	1863
21	354,225,301	169	28	5,584,163,673	2663
22	547,600,561	261	29	7,895,408,601	3764
23	833,501,761	397	30	11,054,221,305	5271

Table 8: Endgame position counts and database sizes for Kalah(6, n).

4. RESULTS

Table 9 provides the exact game values for Kalah with varying numbers of holes and initial counters per hole, expressed in the difference of kalahah content. Moreover, we provide an example of a perfect game from both sides. For each perfect game, holes are indexed from 0 on each side, play proceeds from left to right, and a dash indicates a switch of the player to move. For example, 45-3 means the first player moves from hole 4 and hole 5 by getting an extra turn, and then the next player moves from hole 3.

To solve Kalah(6, 5), we used a Linux machine with an 800 MHz AMD Athlon processor and 256 MB of RAM. 107 MB were used for a 20-counter endgame database, and 96 MB were used for an 8 mega-entry transposition table. With this set-up, it took 4.7 hours to prove the minimax value of Kalah(6, 5), and 35 seconds for Kalah(6, 4).

Again using the same set-up, we also attempted to solve Kalah(6, 6). After 4.2 days, the program had yet to complete its first full depth test call for MTD(f), which would have determined whether or not the first player can win by at least 3 stones. It could take anywhere from a few weeks to several months to complete the full computation.

5. DISCUSSION AND FUTURE RESEARCH

Solving the game of Kalah increases the hope that Awari will also be solved in the near future. The analysis of the game graphs for Kalah suggests that the state-space complexity for Awari might be smaller than the current estimation of 10^{12} as well. Furthermore, the estimation of the game-tree complexity for Awari (10^{35}) might be

$m(n)$	Game value	Perfect game
4(1)	2 (W)	32-32-1
4(2)	6 (W)	3-3-230-231-1323
4(3)	8 (W)	13-1-032-2-3-0-1320-3-0-2
4(4)	2 (W)	02-2-3-0-2-21-303231-0-32-1-0
4(5)	2 (W)	0-2-1-3-0-2-1-0-32-3-021-2-0-02-2-1-3-3-313032-13031-1-2-2-3
4(6)	0 (D)	1-0-2-3-1330-01-2-2-21-201-2-23-3-23-230-32-32-12-1-0-32
5(1)	0 (D)	43-43-2-2
5(2)	0 (D)	31-32-24-0-0-31-3
5(3)	8 (W)	24-3-3-03-41-1-04342-3-3-2-2-4-414342-3
5(4)	12 (W)	12-04-03-2-20-41-1-42-32-3-2-40-0-1
5(5)	2 (W)	03-2-2-1-1-23-24-2-0-34414340-4240-2-14342
5(6)	2 (W)	2-0-0-3-3-0-2-1-440-4-42-2-121-12-3-1-4-20-0-34-24-324-1-3-3-1-41-43-2-2-3
6(1)	2 (W)	54-54-3-3-2
6(2)	10 (W)	42-42-30-0-1-1-4-5
6(3)	2 (W)	4-5-35-250-2-154-451535452-53-3-54-2
6(4)	10 (W)	25-10-3-3-5153-1-4-5-045-4-535452-53-4-1-3-2-0-54-1-3
6(5)	12 (W)	12-02-05-2-4-51-53-3-45-20-3-2-2-345-5-4-351-0-54-1-52-354-4-254-3-3

Table 9: Game values and perfect games for Kalah(m, n).

$m\backslash n$	1	2	3	4	5	6
1	D	L	W	L	W	D
2	W	L	L	L	W	W
3	D	W	W	W	W	L
4	W	W	W	W	W	D
5	D	D	W	W	W	W
6	W	W	W	W	W	

Table 10: Summary of the game values of Kalah(m, n).

much higher than that of Kalah(6, 4) and Kalah(6, 5), but these estimations do not take into account the many transpositions that occur in Kalah and Awari. The gap between Kalah(6, 5) and Awari might be smaller than the estimations suggest. Large endgame databases for Awari (up to 34 stones) have been constructed already and even larger are being built now (Lincke and Marzetta, 2000). In combination with state-of-the-art search enhancements and the extensive use of transposition tables, the databases should enable the teams to solve the game within one or two years.

In order to fill the last blank in our table, Kalah(6, 6), the program is being rewritten distributively using Cilk-5 (Frigo, Leiserson, and Randall, 1998) and the Young Brothers Wait Concept (Feldmann *et al.*, 1990). The initial difficulties in establishing parallelism during tree search should not be a problem here due to the size of the computation, and the absense of strict time constraints. The parallel version will employ shared transposition tables and endgame databases.

Unfortunately, by solving Kalah, Dakon and Awari (soon), only three games of the large group of mancala games will be done. The rules of many mancala games are similar to the rules of these games, but a small change in the rules can render the process of solving a game very difficult. It would be interesting to perform a general analysis of mancala-game rules in order to determine the influence of each of the rules (and combinations of them) on the complexity of the game. This analysis then would point out which mancala games are so different from these three games that they justify the effort to be solved by computer. Some special mancala-game rules turn the game into a “non-minimax” game. To mention three such rules: (1) sometimes, cheating is allowed, and the players have to negotiate on what to do next, (2) in some games the players move simultaneously, (3) sometimes the opponent is allowed to capture during a player’s turn. It is worth to investigate whether these types of rules can be introduced in computer games.

The full-game databases of Kalah will be used in the near future to measure the quality of evaluation functions for Kalah. In turn, Kalah will play a role in experiments on opponent modelling in game search (Donkers, Uiterwijk, and van den Herik, 2000a).

6. REFERENCES

Allis, L. V. (1994). *Searching for Solutions in Games and Artificial Intelligence.* Ph.D. thesis, Rijksuniversiteit Limburg, Maastricht, The Netherlands. ISBN 90–9007488–0.

Allis, L. V., Meulen, M. van der, and Herik, H. J. van den (1991). Databases in Awari. *Heuristic Programming in Artificial Intelligence 2: The Second Computer Olympiad* (eds. D. N. L. Levy and D. F. Beal), pp. 73–86, Ellis Horwood, Chichester, UK. ISBN 0–13–382615–5.

Bell, A. G. (1968). Kalah on Atlas. *Machine Intelligence*, Vol. 3, pp. 181–194. ISSN 0076–2032.

Donkers, H. H. L. M., Uiterwijk, J. W. H. M., and Herik, H. J. van den (2000a). Investigating Probabilistic Opponent-Model Search. *Proceedings JCIS 2000* (ed. P. P. Wang), pp. 982–985. ISBN 0–9643456–9–2.

Donkers, H. H. L. M., Voogt, A. de, and Uiterwijk, J. W. H. M. (2000b). Human versus Machine Problem Solving: Winning Openings in Dakon. Accepted for publication in Board Games studies.

Feldmann, R., Monien, B., Mysliwietz, P., and Vornberger, O. (1990). Distributed Game-tree Search. *Parallel Algorithms for Machine Intelligence and Vision* (eds. V. Kumar, L. N. Kanal, and P. S. Gopalakrishnan). Springer Verlag, New Yorl. ISBN 3–540–97227–7.

Frigo, M., Leiserson, C. E., and Randall, K. H. (1998). The Implementation of the Cilk-5 Multithreaded Language. *ACM SIGPLAN Notices*, Vol. 33, No. 5, pp. 212–223. ISBN 0–89791–987–4. http://www.cs.virginia.edu/pldi98/program.html.

Heinz, E. A. (1998). Extended Futility Pruning. *ICCA Journal*, Vol. 21, No. 2, pp. 75–83. ISSN 0920–234X.

Jenkins, B. (1997). Algorithm Alley. *Dr. Dobb's Journal*, Vol. 22, No. 9, pp. 107–110. ISSN 1044–789X.

Lincke, T. (2000), Awari Endgame Databases. http://wwwjn.inf.ethz.ch/games/awari.

Lincke, T. and Marzetta, A. (2000). Large Databases with Limited Memory Space. *ICGA Journal*, Vol. 23, No. 3, pp. 131–138.

Murray, H. J. R. (1952). *A History of Board Games other than Chess.* Oxford at the Clarendon Press, London, UK. ISBN 0–87817–211–4.

Nievergelt, J., Gasser, R., Maeser, F., and Wirth, C. (1995). All the Needles in a Haystack: Can Exhaustive Search Overcome Combinatorial Chaos? *Lecture Notes in Computer Science*, Vol. 1000, pp. 254–271. ISSN 0302–9743.

Plaat, A., Schaeffer, J., Pijls, W., and Bruin, A. de (1996a). Best-First Fixed-Depth Minimax Algorithms. *Artificial Intelligence*, Vol. 87, Nos. 1–2, pp. 255–293. ISSN 0004–3702.

Plaat, A., Schaeffer, J., Pijls, W., and Bruin, A. de (1996b). Exploiting Graph Properties of Game Trees. *13th National Conference on Artificial Intelligence*, Vol. 1, pp. 234–239, AAAI Press, Menlo Park, CA. ISBN 0–262–51091–X.

Russ, L. (2000). *The complete Mancala Games Book.* Marlow & Company, New York. ISBN 1–56924–683–1.

Schaeffer, J. (1983). The History Heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16–19. ISSN 0920–234X.

Schaeffer, J. (1986). *Experiments in Search and Knowledge.* Ph.D. thesis, University of Waterloo.

Schaeffer, J. (1989). The History Heuristic and the Performance of Alpha-Beta Enhancements. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, pp. 1203–1212. ISSN 0162–8828.

Slagle, J. R. and Dixon, J. K. (1970). Experiments with the M & N Tree-Searching Program. *Communications of the ACM*, Vol. 13, No. 3, pp. 147–154. ISSN 0001–0782.

Zobrist, A. (1970). A New Hashing Method with Application for Game Playing. Technical Report 88, Computer Science Department, The University of Wisconsion, Madison, WI, USA. Reprinted (1990) in *ICCA Journal*, Vol 13, No. 2, pp. 96–73. ISSN 0920-234X.