Solving the Game of Awari using Parallel Retrograde Analysis

John W. Romein and Henri E. Bal

Vrije Universiteit,
Faculty of Sciences, Department of Mathematics and Computer Science,
Amsterdam, The Netherlands

{john,bal}@cs.vu.nl

Abstract

We have solved the game of awari, an ancient African board game that is played worldwide now. The game is a draw when both players play optimally. To solve awari, we computed several databases that can be used jointly to select the best move from any position that can occur in a game. The largest database contains 204 billion entries (178 gigabyte), and is much larger than the largest (endgame) database for any game computed so far. In total, we determined the results for 889 billion *positions. We solved the game on a large computer cluster, using a new parallel search algorithm that optimally uses the available resources (processors, memories, disks, and network).*

In this paper, we show how we solved awari and how we verified the results. We also present new insights into the game of awari.

Keywords: *awari, distributed game databases, distributed search, retrograde analysis, supercomputing*

Introduction

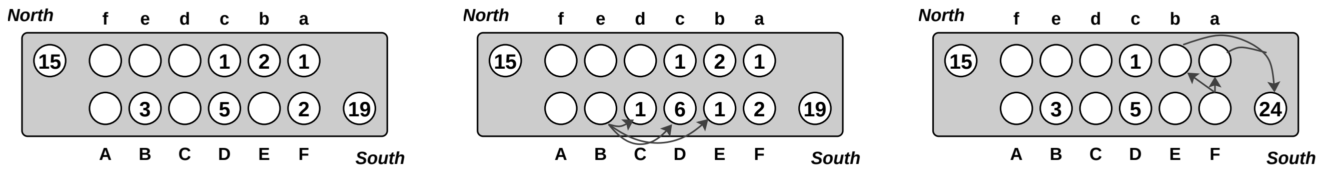
With the progress of the state-of-the-art in processor and network technology, parallel computers can accept computational challenges that were inconceivable before. One such challenge is to solve the game of awari, a 3500-year-old board game that originates from Africa, which is played all over the world now. Being the best-known variant of the class of mancala games (games played on a board with pits and stones that are sown around), awari has drawn much attention from computer science and artificial intelligence researchers [6], who have been studying techniques to play the game for more than a decade [3].

We have solved awari on a modern, parallel computer, using 144 processors, a total of 72 GB main memory, and a fast interconnect (Myrinet). To fully solve the game, we had to determine the score of *889.063.398.406* positions. The score of a position exactly foretells the result at the end of the game when both players play optimally, and can be used to select the best move. The scores are stored in a collection of databases, one for each number of stones. The largest database, which contains scores for each reachable position with all 48 stones, has 204 billion entries and occupies 178 GB. This amply exceeds the largest database for any game computed so far, both in number of positions and in storage size. For several reasons, it is not possible to cut this database into smaller, independent pieces. Also, other attempts to solve the game were unsuccessful (see Section "related work"). No single computer has enough processing power and memory to compute the databases, but even for parallel computers the problem is extremely challenging.

To create the databases, we developed a new parallel algorithm that is based on retrograde analysis (RA). RA starts from all terminal positions in the game, of which the outcomes are trivially known, and reasons "backwards" to the starting position of the game. The algorithm uses the main memories economically for frequently and randomly accessed data, and keeps terabytes of less frequently accessed intermediate results on disks. All processors repeatedly inform each other about positions for which RA has (partially) determined the outcome. This results in a huge amount of interprocessor communication, over a petabit (10^{15} bits) in total, and terabytes of disk I/O. The crux of our algorithm is to make all this communication and disk I/O asynchronous, so processors do not have to wait and can continue doing useful work. This *balanced* and *concurrent* use of resources (processors, memory, disks, and network) leads to a highly efficient algorithm: no single resource is an evident bottleneck. Solving the game took 51 hours, a surprisingly short time thanks to the many optimizations we applied.

We put much effort into verification of the databases. In an environment where we execute quadrillions of instructions, store terabytes of data, and communicate a petabit, we had to deal with possible hardware, operating system, and application faults. Writing a correct and efficient parallel application is difficult, because the desire to be as efficient as possible leads to many opportunities for race conditions and other mistakes.

Below, we explain the rules for awari and discuss previous work. Then, we explain the algorithm we use. Next, we



(a) Position to move from. South is to move.

(b) After move from B. Now North is to move.

(c) Alternative move from F. Stones from b and a are captured.

Figure 1. Example awari move. The numbers in the circles indicate the number of stones in the pits.

elaborate on new insights into the game of awari, and we discuss verification measures.

The awari rules

Awari is played on a board where each player owns 6 pits: A–F for the player called “south”, and a–f for the player called “north” (see Figure 1(a)). Each player also owns an auxiliary pit on the right-hand side of the board, which is used to keep captured stones. In the initial position, all pits (except the capture pits) are filled with four stones, thus the initial position has 48 stones. All stones have the same color. The player to move chooses one of his own nonempty pits and removes all stones from the pit. The player then *sows* the stones counterclockwise over the other pits, one stone into each next pit, and skips the original pit if there are more than 11 stones to sow. This is illustrated by Figure 1(b), where the three stones from B are sown into C, D, and E.

After the sowing phase, stones can be *captured*. If the last stone is sown into an enemy pit that contains 2 or 3 stones (after sowing), the stones are captured. In this case, if the second last pit is also an enemy pit that contains 2 or 3 stones, they are captured as well, and this process is repeated clockwise. Captured stones are moved into the player’s capture pit and remain there during the rest of the game. The player who captures most stones wins the game. Figure 1(c) shows the result after making capture move F from Figure 1(a). The stones are first sown into a and b, and subsequently captured with the other stones from b and a.

When a player cannot move (i.e., all own pits are empty), the remaining stones are captured by the opponent, ending the game. However, to avoid such a situation early in the game, it is not allowed to do a move that leaves the opponent without countermove, unless *all* moves eradicate the opponent. In Figure 1(a), this prohibits a move from D, since it eradicates the opponent and noneradicating moves are available. (The best move is from F: south can win 27–21). Finally, a repeated position results in an even division of the remaining stones, conceptually splitting the last stone into two halves if there is an odd number of stones remaining.

Unfortunately, the rules slightly differ from one country to another. For example, capturing *all* opponent’s pieces in a single move is sometimes allowed. We use the rules that are used by all computer scientists since the first publication on computer-aided play of awari [3].

Related work

Despite the interest for games that computer scientists have shown already since the early days of computing history, games are solved at rare occasions. Disregarding the solving of kalah (a much simpler awari variant) in 2000, it has been a long time since the previous widely-known games were solved (nine-men’s-morris in 1993 [5], gomoku in 1992 [2], and connect-4 in 1988 [1]). Some games were solved analytically, others by brute force, or by a combination of both. A recent article by Van den Herik et al. presents an overview of solved games [6].

So far, research on awari has focused on endgame databases, since they can play a decisive role in a game. In 1995, Bal and Allis used a parallel retrograde analysis program to construct databases of up to 22 stones on a distributed-memory machine [4]. Five years later, van der Goot used a different parallel retrograde analysis variant to build the 40-stone database, on a large shared-memory machine [13].

Lincke and Marzetta published a sequential, memory-limited algorithm to compute awari endgame databases [9]. As of June 2002, a program implementing the algorithm has already run for many months, trying to compute the 40-stone database. The algorithm requires two bits per position on disk (accessed sequentially), one bit per entry in main memory (accessed randomly), and some additional disk space to store intermediate scores and capture information.

With the 40 resp. 39-stone databases available, the latter two groups (independently) tried to solve awari by performing a large forward search from the initial position down to positions that hit the constructed endgame databases, a technique that has been used successfully for games like kalah, nine-men’s-morris (see below), and the sliding-tile puzzle. However, this failed for awari; the forward search tree was too large and did not hit the databases fast enough, since optimal play tends to postpone captures quite long.

Databases have also been constructed for other games. By far the most impressive ones are the checkers endgame databases built by the games group of the University of Alberta [7]. By June 2002, the databases contain all positions with up to 8 pieces and some of the 9 and 10-piece positions, for a total of 3.3 trillion states. This is more than the total number of states for all awari databases, but for several reasons, the awari databases are harder to create. First, the limiting factor is the size of the hardest subproblem (database) that must be solved at once. Awari cannot be split into as many subproblems as checkers can be, and consequently, the largest awari database contains 2.5 times as many entries as the largest checkers database currently solved. Second, the awari databases contain scores between -48 and 48, while the checkers databases only distinguish between win, draw, and loss, and thus occupy far less space. Also, a checkers-specific compression technique reduces the storage requirements of the created databases to only 40 GB. The databases play an important role in Chinook, the undisputed world's strongest checkers-playing program.

Gasser solved nine-men's-morris [5], also using a combined forward and backward search. The backward search resulted in databases that contained 7.67 billion states in total.

Endgame databases for chess have also been built, for subgames with at most 6 pieces on the board [12]. Although they have revealed remarkable new insights, they are more of theoretical interest than of practical purpose, since a chess game is usually decided well before the stage of the 6-piece endgame is entered.

Retrograde analysis

A position contains captured and remaining (noncaptured) stones. Although captured stones do contribute to the final outcome of a position, the best move from a position does not depend on the captured stones. We therefore only consider the distribution of the remaining stones. This significantly reduces the number of states to be searched and stored, since positions that only differ in captured stones are now considered equal.

The *score* of a position depends on how the remaining stones will be divided eventually, after optimal play from both players. The score is the difference in stones after such a division: positive if the player to move will capture most of them, and negative if the other player will capture most. For example, the score of the position in Figure 1(a) is +2, because optimal play on both sides divide the remaining stones 8–6 to south's advantage (since south has already captured four stones more than north, it will win +6).

All reachable positions (where the capture pits are omitted) form a directed, cyclic graph called the *state space*. Figure 2 shows a small part of the state space for awari. The nodes represent positions, and the arcs represent moves. The top node in the figure corresponds to the position in Figure 1(a). Each arc is labeled with the pit name from which the stones are sown, and possibly with the number

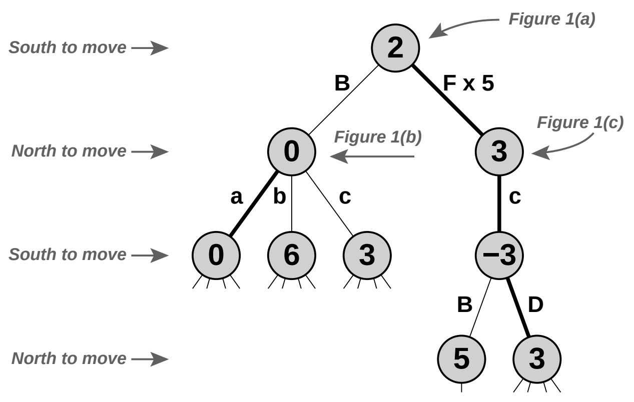


Figure 2. The search of a simple state space.

of stones captured during the move (in the figure, move F from the top node captures 5 stones). The root of the state space is the initial position and the leaves are terminal positions, where no moves are available (neither is shown in the figure).

The numbers in the nodes are the scores. RA determines the scores bottom-up. The scores of the leaves follow from the rules of awari and equal the negated number of stones on the board, since the remaining stones are taken by the opponent. For the interior nodes, the score equals the maximum of the number of stones captured minus the score of the child:

$$Score(p) = \max_{c \,\in\, Children(p)} (CapturedStones(p,c) - Score(c))$$

The scores of the children are negated to reflect the opposed interests of the players. From the top node, move F is better than B, since it results in a score of 2 (5 captured stones minus child score 3) rather than 0. Each best move is shown by a thick line.

The figure shows a tree, but in fact, the state space is a graph, since nodes can (and usually do) have multiple parents. Moreover, the graph is cyclic since there are move sequences that yield repeated positions. Nodes in cycles without outlet have scores of zero, since they will be drawn by repetition.

Our goal is to compute the scores of all nodes, and in particular that of the root node, to determine the game-theoretical score of the entire game, using RA as bottom-up search technique. At each backward step, RA computes the *reverse moves* to discover all parents of a node. When the score of a node is established, the score is reported to its parents. When a parent has received the scores from all its children, its score is established. This process is repeated recursively, until the score of the root is determined. Various refinements for RA exist, often trading memory for computation effort.

Awari databases

Searching the state space at one go is impossible, due to its size. We therefore split the state space into subgraphs that can be searched one after another. The state space is split with respect to the number of noncaptured stones in a position. For each subgraph, we construct a database that contains its scores. Each *n*-stone awari position maps uniquely to a position in the *n*-stone database. Since captures cannot be undone, the databases can be computed one after another, starting with the 0-stone database and ending with the 48-stone database, skipping 47-stone positions since they cannot occur in a game.

n	entries
0	1
1	12
2	78
3	364
4	1,365
5	4,368
6	12,375
	…
41	47,062,945,644
42	58,806,619,443
43	73,127,652,312
44	90,517,649,586
45	111,548,476,480
46	136,883,249,790
48	203,648,015,936

Table 1. Number of entries for some of the *n*-stone databases.

We only store positions where south is to move; if north is to move, the position can be looked up in the database by rotating the board. We also omit the positions where all six pits of the nonmoving player are occupied. Those positions are unreachable (except for the starting position), since the nonmoving player always left at least one of his own pits empty during the previous move. Although hard to implement, omitting unreachable positions results in reductions in computation time and database sizes of up to 27%. Each *n*-stone database contains $\binom{n+11}{11} - \binom{n+5}{11}$ positions (the number of possible stone permutations minus the number of unreachable permutations). Table 1 lists the number of entries for some of the databases.

The databases are indexed by Gödel numbers of the positions [3], which enumerate all positions in a database. We modified the Gödel numbers to take unreachable positions into account. Functions exist to compute the index number from a board and vice versa.

Each database entry stores a score between -48 and +48 and occupies 7 bits. The larger databases do not fit in main memory: the 48-stone database requires 178 GB. Moreover, we need additional main memory during computation of the database to store intermediate results. Unfortunately, it is not possible to split the databases into smaller pieces (e.g., with respect to the number of stones in a particular pit), because interdependencies within a database (cycles in the corresponding subgraph, that mutually influence the entries' values) compel that each database is computed at one go. Another complication is that the indexing scheme often maps parent and child nodes in the state space quite far from each other in the database. Since parents and children often interact, the databases are accessed randomly during construction. Storing the databases on disk and accessing the entries randomly is prohibitively slow. Since we cannot reduce the number of entries in a database, we reduce the number of *bits per entry* stored in main memory, as explained in the next section.

Memory-limited parallel retrograde analysis

Several algorithms for retrograde analysis exist, each with its own memory requirements and efficiency characteristics. Memory-limited algorithms trade memory requirements for increased computational effort, by storing as little as one bit per database entry in main memory, and the remainder on disk [8, 9]. These algorithms avoid random disk accesses and access disk data only linearly, allowing transfers of large amounts of consecutive data. In contrast, data in main memory may be accessed randomly. The choice of the algorithm depends on the availability of main memory. For databases of up to 41 stones, we use the algorithm described by Bal and Allis [4], which is computationally the most efficient, but also the most memory-intensive algorithm, requiring 10 bits per entry in main memory. For the larger databases, we use a new parallel, cost-efficient algorithm that is almost three times slower, but requires five times less memory: each entry requires only 2 bits in main memory. The details of the algorithm are described in the sidebar *(the contents of the sidebar are appended at the end of this article)*.

The algorithm is parallelized by partitioning the databases over the machines. To improve the load balance, the entries are interleaved over the machines rather than partitioned in consecutive chunks (e.g., with 144 processors, CPU 2 stores entries 2, 146, 290, . . .) [4]. The algorithm is difficult to parallelize efficiently, because the children and parents of a node are usually all located at different processors. Performing a *synchronous* remote lookup to check the score of a child or parent is prohibitively expensive, because the processor has to wait for a reply, causing it to idle most of the time.

To avoid high communication and synchronization overheads, we parallelized the search in such a way that communication is highly asynchronous (nonblocking). Rather than shipping *data*, we migrate *work* to other processors. Instead of performing remote lookups, a remote processor must continue the work on a particular node. As soon as a computation requires data from a remote processor, the work itself is transferred to and resumed on the remote processor. Each processor uses a message receive queue that contains work to be processed. As long as there is work in the receive queue, the processor accepts and performs the work. After the search, a global termination detection algorithm is used to check that all work queues are empty and to ensure that no messages are in transit.

This work-migrating scheme has important advantages. First, it keeps the processors busy, since they do not have to wait for (synchronous) reply messages. Second, it allows combining multiple pieces of work into a single, large network packet; this significantly reduces the communication overhead. Third, our algorithm is insensitive to network latency, and fourth, we use the processors and network devices concurrently by overlapping communication and computation. In top-down search, we experienced the same benefits [11]; the scheme is applicable to any parallel graph search algorithm.

Since we can only store 2 bits per entry in main memory, the remaining state is kept on the local disks. We only access the files sequentially to avoid slow disk head seeks. To lower the overhead of disk I/O, we explicitly hint the operating system which parts of the disk databases should be mapped into main memory. We prevent blocking on disk reads by aggressively prefetching data from disk, so that the data are available when they are actually needed. After the data are used and possibly modified, we hint the operating system to lazily write back dirty data and free up memory resources, since these data will not be used for a while.

By properly integrating asynchronous disk I/O with the asynchronous parallel search algorithm, all resources are used concurrently, for optimal performance. The scheme is balanced in such a way that none of the resources is an apparent bottleneck.

Performance results

We computed the databases on a cluster with 72 dual-processor machines, built by IBM. Each machine contains two 1.00 GHz Pentium III processors, 1 GB main memory, and a 20 GB IDE disk. The system is connected through Myrinet, a switched network with 2 Gb/s bidirectional links. The application is implemented in C and uses Myricom's GM communication library. Construction of the 48-stone database required 15.2 hours. It took 51 hours in total to compute all databases.

The application communicates heavily. During the processing phase (where 50% of the time is spent), each SMP node sent and received 20 to 30 MB/s, thus the Myrinet switch routed 1.4 to 2.1 GB/s, excluding protocol overhead. In total, 130 TB (1.0 petabit!) of data were communicated. The amount of disk I/O is estimated to be in the order of ten terabytes.

New awari insights

The newly constructed databases revealed that awari is a draw, so being able to begin the game is neither an advantage, nor a disadvantage. Remarkably, the only good opening move is from the rightmost pit (F); the other opening moves are losing.

Figure 3 shows how the scores of the positions are distributed. The shape of the histogram reveals that most positions are centered around the bar "score = +2". For random positions, the average score is +2.29, indicating an advantage for the player to move. 60.0% of the positions are winning, and 11.3 % are drawn. The different gray shades distinguish the number of stones on the boards. The alternating sizes of a particular number of stones indicate an odd/even effect in which positions with an even number of stones often have an even score and vice versa.

We also observed that capture moves are not always the best moves: for 22% of the positions where one has the choice between capture and noncapture moves, it is better to do a noncapture move.

The databases were used to replay and analyze the games at the Computer Olympiad 2000, where the two strongest awari-playing programs were competing in a tournament [10]. Although it was believed that the programs played close to perfectly, even in the final and decisive game, the eventual winner made 13 wrong moves before starting perfect endplay! In six out of eight games, both programs made mistakes while in their opening books; errors were made as early as at south's third move. Our databases thus significantly overtake the playing strength of the current world champion.

Since we store scores rather than best moves in our databases, it is not always directly clear which move to take if there are multiple moves available that have the same best score. There is a complication: if the score is positive, one should avoid a repeated position; otherwise the remaining pieces are divided and the indicated positive score is not obtained. Occasionally this requires searching the databases, but it is likely that this can always be done quickly [8]. Assuming this is true, we have *strongly* solved the game of awari (according to the classification of Allis [2]), meaning that for all positions that can occur in a game, a strategy is known to obtain the game-theoretical score of the position, under reasonable resource constraints.

Our databases are accessible through a web server (`http://awari.cs.vu.nl/`). A Java applet with a graphical interface can be used to look up positions, showing the scores for each possible move. The applet can also be used to play a game, at an adjustable playing level.

Database verification

Hardware and software bugs could silently corrupt the databases during construction. The following precautions minimize the chance that such errors occur.

The hardware has some facilities to detect and correct faults. The main memories, processor caches, and the memories on the Myrinet cards use error correcting codes (ECC). Moreover, the GM network software handles CRC errors for packets that are corrupted on the fiber cables. The operating system detects and handles (nonfatal) CRC errors that occur during data transport from and to the disks.

We put much effort in verification of the created databases, since writing a correct parallel program is a difficult task, especially when global termination detection is involved. We therefore took the following measures:

- We made sure that the different retrograde search implementations (those using 2 resp. 10 bits per entry) created exactly the same databases for up to 41 stones.

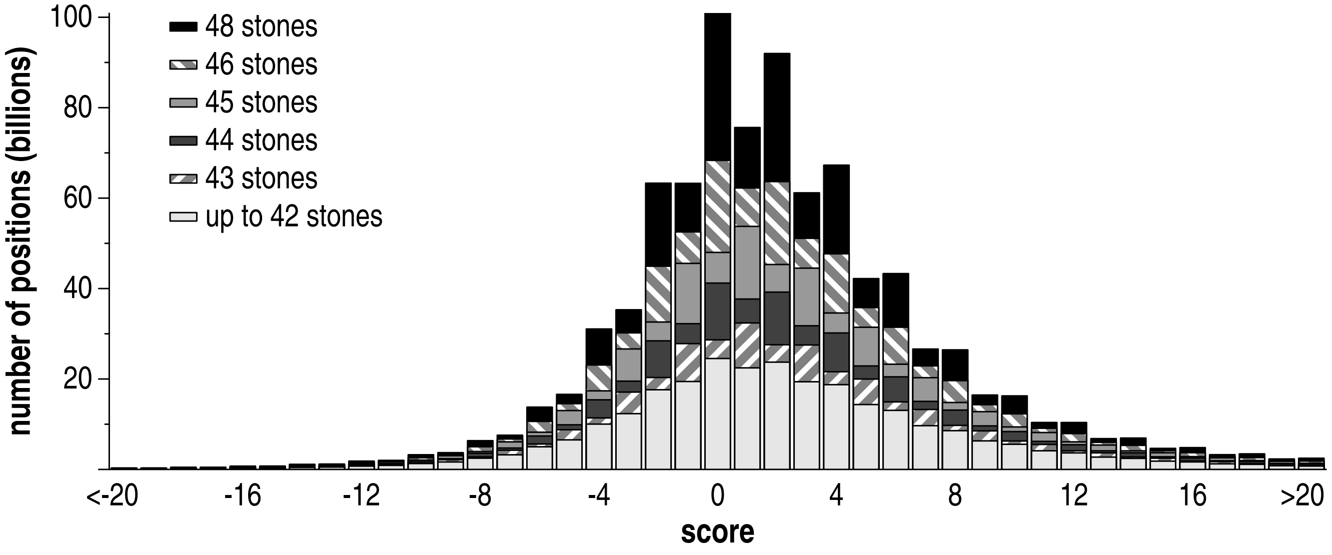


Figure 3. Distribution of the positions' scores.

- We wrote a separate program that checks whether the parent and children values in the databases are consistent.
- We recomputed all databases on a different number of processors (64 dual SMP nodes) and did a bitwise comparison with those generated by the 72 SMP nodes, which showed that all entries were exactly the same. Smaller databases were recomputed over and over again, also using different numbers of processors.

This made us confident that the programs are free from race conditions, but it does not protect us against (sequential) bugs in the indexing functions and the (un)move generators that generate the children and parents of a node. The indexing functions are complex due to the omission of unreachable positions, and generating parents is intricate due to awari's complex capturing rules. We therefore also compared our results to those by Lincke and Marzetta (of up to 36 stones)[1]: the results were the same.

Conclusions and future milestones

We solved the game of awari, and determined that the game is a draw when both players play optimally. We determined the scores for 889,063,398,406 positions, enumerating all states that can possibly occur in a game. Due to the size of the 48-stone database and the random way the databases are accessed during construction, this could not have been achieved without a large computer system with a well-balanced algorithm. Construction of the databases took 51 hours on a 144-processor 1 GHz Pentium III cluster, equipped with 72 GB main memory and a 2 Gb/s network.

[1]To make the results comparable, we constructed separate databases that *included* unreachable positions.

We used a retrograde analysis algorithm that requires two bits per entry in main memory; the remaining data were kept on disk. The parallel algorithm is highly asynchronous, resulting in an application that profits from low communication overhead, no idle processors, network latency hiding, and overlap in computation, communication, and disk I/O. We also implemented many optimizations to decrease the execution times.

We put much effort into verification of the databases, to make sure that they are correct. We compared (partial) results from two algorithms, different numbers of processors, and results obtained by other researchers, and verified everything using a separate program that checked the database consistency.

After this milestone, the question arises which game will be solved next, and when (see an article by van den Herik et al. for a discussion [6]). We expect that checkers will be the first game solved, possibly within the current decade. Although its state-space complexity is six orders of magnitude larger than awari's (estimated at 10^{18} [2]), a combined top-down and bottom-up search is likely to work here (because the game contains many forced captures), and renders it unnecessary to search the entire state space. Othello will be harder to solve, and chess will definitely not be solved in the foreseeable future, at least not using brute force alone. Even though databases for these games require fewer bits per state and can be split into more disjoint subproblems than awari, their state-space complexities are too large (estimated at 10^{28} and 10^{46}, respectively [6]).

The awari databases, more statistics, and an (infallible) awari-playing program are accessible via `http://awari.cs.vu.nl/`.

References

[1] J. D. Allen. A Note on the Computer Solution of Connect-Four. In D. N. L. Levy and D. F. Beal, editors, *Heuristic Programming in Artificial Intelligence 1: the First Computer Olympiad*, pages 134–135. Ellis Horwood, Chichester, England, 1989.

[2] L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, the Netherlands, September 1994.

[3] L. V. Allis, M. van der Meulen, and H. J. van den Herik. Databases in Awari. In D. N. L. Levy and D. F. Beal, editors, *Heuristic Programming in Artificial Intelligence 2: the Second Computer Olympiad*, pages 73–86. Ellis Horwood, Chichester, England, 1991.

[4] H. E. Bal and L. V. Allis. Parallel Retrograde Analysis on a Distributed System. In *Supercomputing '95*, San Diego, CA, December 1995.

[5] R. Gasser. Solving Nine-Man's-Morris. *Computational Intelligence*, 12(1):24–41, 1996.

[6] H. J. van den Herik, J. W. H. M. Uiterwijk, and J. van Rijswijck. Games Solved: Now and in the Future. *Artificial Intelligence*, 134(1–2):277–311, January 2002.

[7] R. Lake, J. Schaeffer, and P. Lu. Solving Large Retrograde Analysis Problems Using a Network of Workstations. In H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk, editors, *Advances in Computer Chess 7*, pages 135–162. Universiteit Maastricht, the Netherlands, 1994.

[8] T. R. Lincke. *Exploring the Computational Limits of Large Exhaustive Search Problems*. PhD thesis, ETH Zurich, Swiss, June 2002.

[9] T. R. Lincke and A. Marzetta. Large Endgame Databases with Limited Memory Space. *Journal of the International Computer Games Association*, 23(3):131–138, September 2000.

[10] J. W. Romein and H. E. Bal. Awari is Solved (note). *Journal of the International Computer Games Association*, 25(3):162–165, September 2002.

[11] J. W. Romein, H. E. Bal, J. Schaeffer, and A. Plaat. A Performance Analysis of Transposition-Table-Driven Scheduling in Distributed Search. *IEEE Transactions on Parallel and Distributed Systems*, 13(5):447–459, May 2002.

[12] K. Thompson. Retrograde Analysis of Certain Endgames. *Journal of the International Computer Chess Association*, 9(3):131–139, September 1986.

[13] R. van der Goot. Awari Endgame Databases. In *Computers and Games*, volume 2063 of *Lecture Notes in Computer Science*, pages 87–95. Springer, 2002.

Appendix

2-bit parallel retrograde analysis

Our parallel RA algorithm requires 2 bits per entry in main memory, sufficiently few to compute the 48-stone database on our machine. It is inspired by the sequential algorithm from Lincke and Marzetta [9], but modified to take advantage of having two bits per entry in main memory instead of one, saving much redundant work and many disk accesses.

```
ENUM State = (Win, DrawOrWin, Loss, Unknown);
VAR States : ARRAY [1 .. NrEntries] OF State;
VAR Scores : ARRAY [1 .. NrEntries] OF [−48 .. 48];
VAR Bound  : [1 .. 48];

PROCEDURE DoCaptures(Index, Board)
    Best := −48;
    IF HasCapturesMoves(Board) THEN
        Best := BestCaptureScore(Board);
    IF Best >= Bound THEN
        States [Index] := Win;
    ELSE IF Best > −Bound THEN
        States [Index] := DrawOrWin;
    ELSE IF NOT HasNonCaptureMoves(Board) THEN
        States [Index] := Loss;
        FOR Parent IN Parents(Board) DO
            States [BoardToIndex(Parent)] := Win;
END;

PROCEDURE PreProcess()
    FOR Index IN 1 .. NrEntries DO
        IF States [Index] = Unknown THEN
            DoCaptures(Index, IndexToBoard(Index));
END;

PROCEDURE DoNonCaptures(Index, Board)
    IF States [Index] = Unknown AND
        AllNonCaptureMovesAreWinsForOpponent(Board) THEN
        States [Index] = Loss;
        FOR Parent IN NonCaptureParents(Board) DO
            ParentIndex := BoardToIndex(Parent);
            IF States [ParentIndex] <> Win THEN
                States [ParentIndex] := Win;
                FOR Gp IN NonCaptureParents(Parent) DO
                    DoNonCaptures(BoardToIndex(Gp), Gp);
END;

PROCEDURE Process()
    FOR Index IN 1 .. NrEntries DO
        DoNonCaptures(Index, IndexToBoard(Index));
END;

PROCEDURE PostProcess()
    FOR Index IN 1 .. NrEntries DO
        CASE States [Index] IS
            Win       : INC(Scores [Index]);
                        States [Index] := Unknown;
            DrawOrWin : (* nothing *)
            Loss      : DEC(Scores [Index]);
                        States [Index] := Unknown;
            Unknown   : States [Index] := DrawOrWin;
END;

PROCEDURE RetrogradeAnalysis(NrStones)
    FOR Bound IN 1 .. NrStones DO
        PreProcess();
        Process();
        PostProcess();
END;
```

Figure 4. Retrograde search algorithm for 2 bits per entry in main memory.

Sequentially, the algorithm roughly works as shown in Figure 4. The procedure RetrogradeAnalysis computes the scores in the NrStones-database (thus multiple invocations are needed to generate all databases). The function repeatedly determines for all n-stone positions whether their values are in a window (-Bound, Bound). By trying multi-

iteration	window	positions
1.	(-1, 1)	W L W D W L L …
2.	(-2, 2)	D L D D W D L …
3.	(-3, 3)	D D D D W D L …
total score →		1 -2 1 0 3 -1 -3 …

Figure 5. Determination of the scores for a 3-stone database.

ple windows, the scores can be derived indirectly. The initial window is (-1, 1); after each iteration, the window is widened. A position is called a "win" if its value is at least Bound, a loss if it is at most -Bound, and a "draw" otherwise. For example, if the algorithm determines that a position is a "win" in iteration 5, it has derived that the position's score is at least 5.

At the end of each iteration, each "win" increments the position's score by 1 and each "loss" decrements the position's score by 1. This is illustrated by Figure 5, where we horizontally list for each position whether it is a "win" (W), "draw" (D), or "loss" (L) after a particular iteration. The scores are accumulated vertically. For example, if this value is 5, the position is counted as a "win" during the first five iterations, and a "draw" during the remaining iterations.

The algorithm in Figure 4 keeps an array States in main memory and an array Scores on disk. The former is accessed randomly and the latter sequentially. Entries in States require two bits and are one of Win, DrawOrWin, Loss, or Unknown. Scores accumulates the position's scores and will eventually contain the final results. Each entry requires 7 bits to store a value between -48 and 48. Initially, all entries in States are Unknown and all values in Scores are 0.

During each iteration, three processing phases are required to determine the win/draw/loss value of each position. The *preprocessing* phase deals with capture moves and lost positions. For brevity, we do not discuss the details, but the idea is that we initialize the entries in States from capture moves, for which the scores are found in previously computed databases,[2] and from terminal positions (in awari, all terminal positions are lost positions).

The *processing* phase performs a scan over all database entries. An Unknown state for which all children are Wins, is assigned a Loss (no matter which move is made, the opponent will win, thus the position is lost). Moreover, all parents are assigned a Win, since the opponent has a winning move available from those positions. Next, the grandparents undergo the same test, recursively. This is depicted in Figure 6 (the numbers and fat arrows in the figure are referred to later). The position marked U→L is initially Unknown and its children are Wins, thus the position itself is assigned a Loss. Then, the parents that are not already Wins

[2]Actually, BestCaptureScore uses a precomputed, auxiliary database that keeps values for capture moves (as suggested by Lincke and Marzetta [9]) to avoid repeated accesses to the smaller databases.

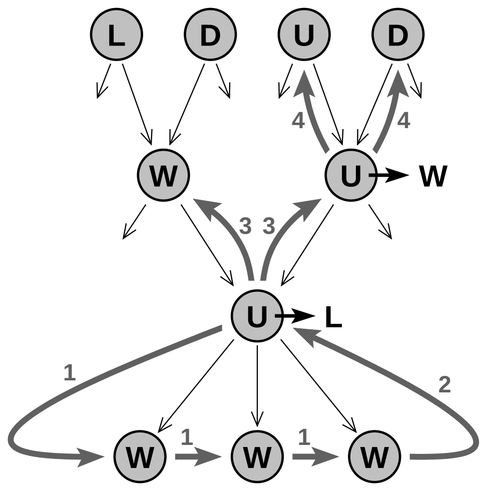


Figure 6. Actions performed during the processing phase.

become Wins now, and the grandparents recurse into the same process. For a better understanding of the algorithm, it is important to see that all Wins are determined through backpropagation of lost positions. Also, DrawOrWins can become Wins, but Losses and Wins retain their values during this phase. Unlike Lincke and Marzetta's algorithm, the processing phase does a *single* scan over all database entries, since it scans parents of entries that have just been set to Win immediately.

The *postprocessing* phase adds the Wins and the Losses to the Scores; entries that are still Unknown or DrawOrWin are effectively a draw. Although all iterations are in fact independent, we perform them with an increasing window size; this allows us to apply some optimizations when we prepare States for the next iteration: drawn positions remain drawn during the rest of the computation, and Wins and Losses are converted to Unknown. A win can become a win or a draw after the next iteration, and a loss can become a loss or a draw.

We parallelized the algorithm described above in the asynchronous way that is described in the main text. Finding out whether all children are wins or setting all parents to a win requires communication between several processors. This is illustrated by the fat arrows in Figure 6. The processor owning the node marked U→L generates the first child and sends a message to the processor owning the first child (this is message type "1" in the figure). The receiving processor looks up the database entry for the first child, and, if the entry denotes a win, incrementally generates the second child and sends the child to the processor containing the second child (again using message type "1"). If the entry denotes something other than a win, the message is dropped. When no more children can be generated, the processor owning the original state is sent a message that the state is a loss (2). Upon receipt, the processor updates the corresponding database entry, and sends messages to the processors owning its parents, that the parents are wins (3). When a processor receives such a message, it looks up the parent to see if it is already a win. If it is not, it makes the entry a win, and sends messages to the grandparents to check for unknown states (4), recursing the entire process. If the message receive queue of the processor contains work, the work is processed, otherwise the processor resumes the main loop in the procedure Process.