

Mancala in Java: an Experiment in Artificial Intelligence and Game Playing

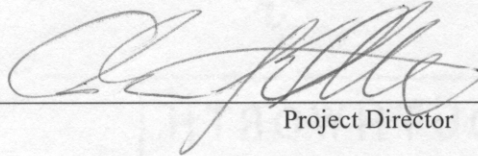
by
Adam Cofer

Departmental Honors Thesis
The University of Tennessee at Chattanooga
Department of Computer Science

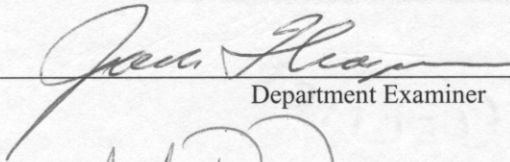
Project Director: Dr. Andy Novobilski
Examination Date: 01 April 2003

Dr. Joe Dumas
Dr. Jack Thompson
Dr. William Lee

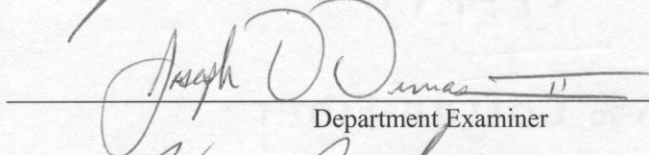
Examining Committee Signatures:



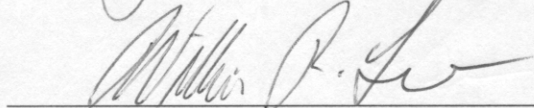
Project Director



Department Examiner



Department Examiner



Liaison, Departmental Honors Committee



Chairperson, University Departmental Honors Committee

Introduction

Mancala is an African board game of prehistoric origin with innumerable variants known by several names, including *pallanguli* (Sri Lanka and southern India), *wari* (Ghana), *awari* (Antilles), *chisolo* (Zimbabwe), *ba-awa* (Ghana), *lontu-holo* (Surinam), and more [1]. Another article linked at [1] on the game states that two researchers at the Free University in Amsterdam, Netherlands solved the game entirely by brute force search methods in May of 2002, but no further material on this recent development could be found.

Previous efforts notwithstanding, the game of mancala provides a rewarding exercise in game theory and artificial intelligence. The goal of this project was to program a satisfactory opponent for mancala, requiring understanding of the techniques of evaluating two-person, perfect information, zero-sum game strategies, necessitating research and careful consideration (game theory terms shall be clarified later, in the section “Move Evaluation Logic and Theory”). Additionally, the project offered flexibility, as creating a novice-level opponent is relatively easy for a game of such simple rules and computational complexity – for perspective, tic-tac-toe requires at most $9!$ move evaluations, whereas checkers requires some 10^{20} evaluations [2]. Given a rate of 680,000 gamestates per second (see “Analysis of Performance” below), tic-tac-toe will require 0.53 seconds at the beginning of the game to find the best move, while checkers will require 4,663,204 years. Mancala, with a possibility of at most 6 moves (and quite often less) at every turn and having an average game lasting approximately 30 moves, will take at most $3.25e17$ seconds, or $1.03e10$ years

to evaluate every possible game state resulting from the first move. To see why it might be possible to evaluate the game fully in a more reasonable time, consider that the evaluation rate can be improved by many means (by implementing the program for specific hardware, or by parallelization of the search across several processors), and that the number of possible moves is usually three or less near the end of the game – meaning that the first half of the game might require less than 6^{15} evaluations while the second half might require 3^{15} , meaning $6^{15} + 3^{15}$ or $4.7e11$ evaluations, about 192 hours' worth. Other techniques can be used to reduce this further, including pruning, parallelization, and depth cutoff with a good evaluation function, explained below.

Rules of Mancala

Play takes place on a board with a large pit at either end and two rows of six smaller pits or bins between them, lengthwise. Each large pit is called a *mancala* or *kalaha* and “belongs” to the player for whom it is on the right. The row of smaller pits on each player’s side belongs to that player. Each of the small pits initially contains four (or some other small number, but four is the most common) stones. The object of the game is for each player to attempt to move as many stones as possible into his or her own *mancala*: at each turn, the player selects a small pit from her row. All stones from that pit are distributed, one by one and counter-clockwise, into the smaller pits and that player’s *mancala* (but skipping the other player’s *mancala*). If the last stone “lands” in the player’s own *mancala*, that player immediately receives

another turn. If the last stone lands in an empty pit on the player's side, all the stones from the opposite pit plus the "capturing" stone are moved to the player's mancala. The game is over when one player has no stones left in any pit. All of the other player's stones are moved to her *mancala*, and the player with the most stones in their *mancala* wins. These rules are written in accordance with the most frequently occurring variation [3].

Design of the Program

Following Object-Oriented design principles, the program is divided into four logical modules, each assigned a representation of an aspect of the game and with relevant functionality: the board itself, representing a state of the game, a node holding a board with links for building a tree of game states, the user interface, and a server, in this case the class holding the main method of the program and the function for finding the best move.

The design process consisted of breaking the simulation problem up into these four parts, then assigning responsibilities to each part for maintaining data and functionality, with the bulk of the work in designing the `findBestMove()` method of the `MancalaServer` class. A listing of the code appears in Appendix I.

Move Evaluation Logic and Theory

Game Theory

“About forty years ago, a mathematician, John von Neumann, and an economist, Oskar Morgenstern, tried to find a more effective way of solving certain kinds of economic problems. They noticed that “the typical problems of economic behavior become strictly identical with the mathematical notions of suitable games of strategy” and so they devised a “Theory of Games.” Oddly enough, this new tool turned out to be invaluable in attacking problems in many other areas as well [4].”

“The theory of games is a theory of decision making [4].” Game theory now has applications far beyond the initial focus of economics, and is an active field of study in sociology, military strategy, artificial intelligence, and financial planning. However, these are areas of application, with much difference between practice and theory: to the player, chess is highly complicated, deep and profound. To the mathematical game theorist, chess is simple – there are a finite number of positions (and thus outcomes), with an easily assigned value for each player at the outcome: win, lose, or draw [4]. Mathematical game theory is additionally restricted by refusing to consider specific aspects of applications, or *how* to evaluate moves or game states. “Generally speaking, criterion-trouble is the problem of what to measure and how to base behavior on the measurements. Game Theory has nothing to say on the first topic, but it advocates a very explicit and definite behavior-pattern based on the measurements [5].”

Games such as chess – and mancala – are classified clearly by game theorists.

We are concerned with mancala: it is two-player – note that in one player, or indeed in n -player games, an element of chance might be considered as a player, often called “Nature”. Mancala is a game of perfect information, wherein both players can observe the full state of the game at all times. Mancala is a zero-sum game; at the end of the game, whatever is won by one player (positive) is lost by the other (negative), and the sum of points (or utility, in economic theory) never changes from zero. Also, it is finite – there is a well-defined end for the game, and it is always reached (a mathematical proof of this will not be offered here, but the reader should be able to satisfy herself of this by considering that once a stone enters a *mancala*, it never leaves, and there are a finite number of stones).

Additionally, assumptions exist about the expected behavior of each player of the game – specifically, that both players are rational, and will always choose the best move. Indeed, “...the sensible object of the player is to gain as much from the game as he can, safely, in the face of a skillful opponent who is pursuing an antithetical goal [5].” Given a set of strategies for player B , player A will assume in making a choice between strategies that B will choose the optimum strategy for herself. Or, in other words, “... we have computed strategies for the personal player as if he were playing against an opponent who shared his valuation of the payoff matrix and who was capable of making intelligent countermoves. This led the personal player to overconservative strategies; he failed to win as much as was available when playing against ironclad stupidity [5].”

A lay person's description of concerns over this apparent shortcoming of game

theory is available [6], wherein it is mentioned that many theorems of game theory are based upon the rational participant in the second party (even more implicit is the assumption that the first party is rational, which one hopes is never in doubt).

The mathematics of n-person game theory stems largely from this construction: for each ply (a *ply* is one move by one player, or half a turn, however, here the words *ply* and *turn* shall be used interchangeably), we may form a game matrix, wherein each row represents a move for one player, and each column, the response. The value located at the intersection is the *payoff* for that move [5]. This can be further complicated: for most game matrices considered in game theory, the value at the intersection of row and strategy columns is the *average* expected payoff for the combination. This average may be weighted in several ways, for example, according to the probability of its appearance [5]. This complication results from non-recursion, and as such does not affect this implementation of a mancala opponent.

The problem of choosing a strategy using these matrices can, of course, be represented as a set of linear functions, and techniques of linear algebra then become available to the problem-solver. This is referred to as *linear programming*, and carries a different sense than its counterpart term in computer science. In mathematical game theory, linear programming is defined as follows: “The problem of maximizing (or minimizing) a linear function (called the objective function) subject to linear constraints. In the most common case, the constraints are loose inequalities, and the variables are restricted to nonnegative values [7].” Further, “...to

each linear-programming problem there is associated a two-person zero-sum game, and conversely, such that whenever the linear-programming problem is soluble the solution to one problem can always be interpreted as providing the solution to the other; second, the proofs of the principal results of both theories use the same formal mathematical tools... [7]” The problem of choosing an optimal strategy – one with the highest possible payoff – in either form always has a solution; this is a result of the proof of the *minimax* theorem, of which there are several. Among the most famous are Nash's [8] and von Neumann's, which was the first, and the theory is given historical treatment in Kuhn's 1952 survey of the literature on the topic. Worth mentioning here is the simplex method of strategy choice, a slight variation on minimax proposed by Dantzig in 1951 [8].

Formal exploration of mathematical game theory can be found in [9] (considered a foundational work in the theory), and [10].

A.I.

The practical application of these ideas to an actual game is, on the surface, quite different from the offerings of theory. The foregoing serves as an historical overview of the development of game theory, and hopefully a small explanation for the origins of practical techniques in artificial intelligence for finite two-person games.

The technique relevant to the implementation of mancala for this project is minimax tree search. For a mathematical explanation, one can again go to [4], but for

a practical explanation [11] is much more helpful. Essentially and in brief, for the reader can refer to the sources cited or to the implementation in Appendix I below for insight, an inverted tree representing the game is built in which the nodes represent game states, and the arcs (or links between nodes) represent moves, or the transitions between states. Note that in this project, the tree is represented in computer memory by a structure built of instances of the MancalaNode class object. Each node is expanded until an endgame is reached, at which point the state is evaluated for a value – in this case, the difference between the number of stones in each *mancala*. This value is consistently positive if one player wins, negative if the other. At each level of the tree or turn, we can expect that one player wants to maximize the value, while at the next level, the other player wishes to minimize it (i.e. maximize it for herself). Thus, the values are promoted up the tree until the root node is reached, at which point the arc with the maximum value is chosen.

It is possible to refine this *minimax* method considerably. Upon observation, many paths through the tree do not need to be taken. If, for example, on the maximizing player's turn move 2 will result in a value of 4, and it is found that move 3 can result in a force to a value of less than or equal to 4 by the minimizing player, it is no longer necessary to evaluate any branches or subtrees descending from move 3. This is known as *alpha-beta pruning* for traditional elements of the algorithm, and again for example please examine `findBestMove()` in Appendix I, or [11].

The mathematical literature notes a phenomenon which appears to correlate to alpha-beta pruning – “domination [4].” If a row or column's values in a strategy

matrix are strictly higher than any other row or column, respectively, then that strategy is said to *dominate* the other, and the lesser strategy need be considered no further.

These methods suffice when there are infinite resources, and in fact would result in perfect play, but what if there is not time or space in memory to build the entire tree? This issue is addressed by evaluation functions at depth cutoff (i.e. at a certain depth, endgame or not, a value is assigned to the game state – this is the solution chosen by this implementation), by iterative deepening of the tree search, by distributed or parallel computing, and by creating books of opening and closing moves referred to in a fast database lookup, or a mixture of the above. For an example of projects which use all of these see [12], and for documentation and theory see [13], [14], [15], and [16].

Finally, the issue of the optimality of minimax when playing against an opponent who does *not* choose the minimum-valued move at each level – an “imperfect” player – must be addressed:

“...Rapoport's conclusion that any deviation from the minimax “can only be of disadvantage” leads to a curious paradox. To see why, observe two basic facts.

Note first that if *either* player selects the minimax strategy, the outcome will be the same: an average gain of five sevenths for player *I*. There is only one way to obtain another outcome: *both* players must deviate from their minimax strategies.

The second point is that the game is zero-sum. A player cannot win unless he or she wins from the opponent; a player cannot lose unless there is a corresponding gain for the opponent.

Putting these two facts together, we have the contradiction. If a player deviates from the minimax, it cannot possibly be “of disadvantage” unless his or her opponent deviates as well. And, by the same argument, the opponent's deviation must also be of disadvantage. But the game is zero-sum: both players cannot lose simultaneously. Obviously, something is wrong.

Here is the answer. There is a strategy for player *I* that guarantees him or her five-sevenths, and the player can be stopped from getting any more; moreover, player *II* is motivated to stop player *I* from getting more. If player *I* chooses another strategy, he or she is gambling. If player *II* also gambles, there is no telling what will happen. The minimax strategies are attractive in that they offer security; but the appeal of security is a matter of personal taste [4].”

Therefore, it is entirely possible, if the human can exploit the shortsightedness of the computer (effectively, to make it 'gamble'), to force the computer into a losing line of play. The strongest bulwark against this is the evaluation function, considering the exponential futility of fully evaluating most games.

Analysis of Performance

What follows is a brief examination of the performance of this project, and a consideration of aspects of tree-based game evaluation as they relate here. All evaluations were conducted on Mandrake Linux 9.1 running Sun JRE 1.4.1 on an AMD 2200+ (1.8 GHz) with 1 GB DDR SDRAM and a 266 MHz FSB.

Time (in seconds)	Nodes Created	Nodes per Second
14.7	10000000	680272.1088
30.2	20000000	662251.6556
43.6	30000000	688073.3945
58.1	40000000	688468.1583
72.6	50000000	688705.2342
97.7	60000000	614124.8721
102.1	70000000	685602.3506
116.6	80000000	686106.3465
131.2	90000000	685975.6098
145.6	100000000	686813.1868
		Average: 676639.3

Figure 1: Nodes per Second

```

bobo@bobo.anderson1.tn.comcast.net: /mnt/data/Projects/dhon
Alpha-beta cutoffs: 150771302
Nodes created: 990000000
Nodes at each depth level:
0:1, 1:1, 2:1, 3:1, 4:1, 5:1, 6:1, 7:1, 8:1, 9:1, 10:1, 11:2, 12:1, 13:2, 14:3, 15:7, 16:25, 17:61, 18:217, 19:498, 20:1736
, 21:4728, 22:11500, 23:24923, 24:49719, 25:94345, 26:174175, 27:316929, 28:565903, 29:1007716, 30:1748009, 31:2986178, 32:
4916881, 33:7867973, 34:11984822, 35:17584879, 36:24339135, 37:31786729, 38:38904397, 39:44173881, 40:46998180, 41:46458088
, 42:43337670, 43:38391148, 44:33053075, 45:28494354, 46:24531939, 47:21425727, 48:19065228, 49:17447037, 50:16500208, 51:1
6337435, 52:17300107, 53:19124509, 54:21706866, 55:24430217, 56:27141955, 57:29788894, 58:31871877, 59:33086156, 60:3294475
4, 61:31874780, 62:29949243, 63:27425606, 64:23921255, 65:20505978, 66:17446808, 67:15025808, 68:13140724, 69:10930129, 70:
8598962, 71:5841886, 72:3776115, 73:1946567, 74:994434, 75:436977, 76:155295, 77:41405, 78:5453, 79:1485, 80:246, 81:36, 82
:0, 83:0, 84:0, 85:0, 86:0, 87:0, 88:0, 89:0, 90:0, 91:0, 92:0, 93:0, 94:0, 95:0, 96:0, 97:0, 98:0, 99:0,
Alpha-beta cutoffs: 152574601
Nodes created: 1000000000
Nodes at each depth level:
0:1, 1:1, 2:1, 3:1, 4:1, 5:1, 6:1, 7:1, 8:1, 9:1, 10:1, 11:2, 12:1, 13:2, 14:3, 15:7, 16:25, 17:61, 18:217, 19:498, 20:1736
, 21:4728, 22:11500, 23:24923, 24:49719, 25:94345, 26:174175, 27:316929, 28:565903, 29:1007716, 30:1748009, 31:2986178, 32:
4916881, 33:7867973, 34:11984823, 35:17584881, 36:24339141, 37:31786749, 38:38904462, 39:44174064, 40:46998635, 41:46459211
, 42:43340010, 43:38395923, 44:33062121, 45:28511367, 46:24563045, 47:21479236, 48:19154786, 49:17589536, 50:16726849, 51:1
6676770, 52:17793105, 53:19780279, 54:22521357, 55:25335701, 56:28062394, 57:30691576, 58:32711549, 59:33809183, 60:3353097
1, 61:32345173, 62:30361341, 63:27788188, 64:24223190, 65:20744929, 66:17616784, 67:15139881, 68:13213001, 69:10970312, 70:
8621567, 71:5850894, 72:3781241, 73:1948047, 74:995158, 75:437085, 76:155319, 77:41405, 78:5453, 79:1485, 80:246, 81:36, 82
:0, 83:0, 84:0, 85:0, 86:0, 87:0, 88:0, 89:0, 90:0, 91:0, 92:0, 93:0, 94:0, 95:0, 96:0, 97:0, 98:0, 99:0,
Alpha-beta cutoffs: 154314275
  
```

Figure 2: Nodes Created at Each Ply, or Depth Level

In Figure 2, note that after 1,000,000,000 nodes have been created in 24.5 minutes, only one of the possible moves at the 13th level has been evaluated. Only at the 15th level has an entire game state possibly been evaluated – and there are billions

left. Therefore, it can be seen why depth cutoff is necessary. Through trial and error, it was determined that the optimal cutoff was at a depth of 12 – this allows the program to return a move within ten seconds.

Depth Cutoff	Time in Seconds				
	Move 0	Move 1	Move 3	Move 4	Move 5
7	0.102	0.155	0.106	0.081	0.083
8	0.318	0.334	0.256	0.201	0.213
9	0.898	0.986	0.695	0.586	0.744
10	2.656	2.889	2.22	1.663	2.221
11	6.302	9.706	6.297	5.06	6.81
12	21.648	25.547	21.169	16.018	20.76
13	53.788	77.302	58.592	49.894	67.528

Figure 3: Time Required for Evaluation as a Function of Depth Cutoff

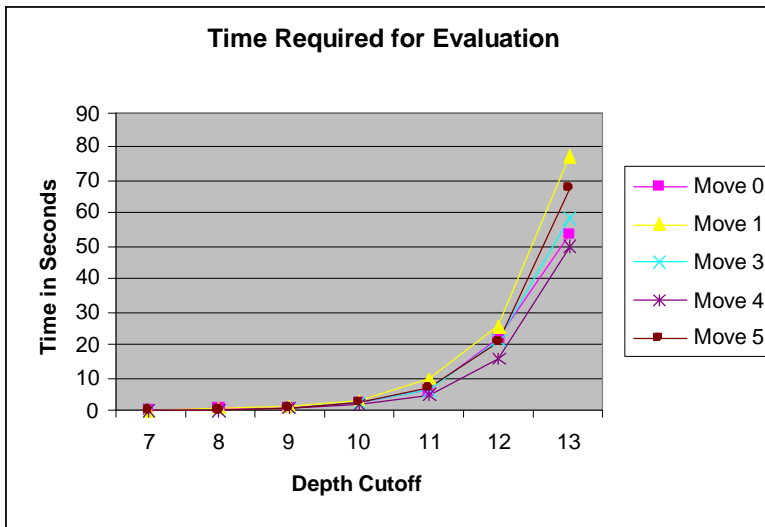


Figure 4: Line Chart of Data in Figure 3

Figures 3 and 4 illustrate what intuition and mathematics will tell us when considering the temporal effect of the depth cutoff parameter (and why 11 was chosen as the depth cutoff for this project, given the acceptable waiting period for a human player). The time required for evaluation increases exponentially as a function of depth.

```

bob@bobo.anderson1.tn.comcast.net: /mnt/data/Projects/dhon
  4  4  4  4  4  4
  [ 0][ 1][ 2][ 3][ 4][ 5]
Player 0 enter a move or type 'quit' to exit:
4
Evaluating moves...
-.....-.....-.....-.....-.....-.....
NodesCreated: 44048900 in 49.894 seconds.
Moving from pit 9.
Evaluating moves...
-.....-.....-.....-.....-.....
NodesCreated: 23297460 in 26.422 seconds.
Moving from pit 8.
Evaluating moves...
-.....-.....-.....-.....-.....
NodesCreated: 32449916 in 36.839 seconds.
Moving from pit 10.

  [12][11][10][ 9][ 8][ 7]
   7  7  0  1  0  5
3  -----  1
   5  5  5  4  0  5
  [ 0][ 1][ 2][ 3][ 4][ 5]
Player 0 enter a move or type 'quit' to exit:

```

Figure 5: Output from Evaluation

In this screen capture, we can observe the effects of alpha-beta pruning on the performance of the algorithm. Each ‘-’ represents a fully evaluated move at the first depth level – all of the moves available to the computer on this turn. Each ‘.’ represents the creation of 1,000,000 nodes in memory. Notice that the last two evaluated moves result in the creation of far fewer nodes; this is a result of pruning as evaluation proceeds from left to right in the tree. For further analysis of the benefits of alpha-beta pruning, see [17].

Finally, the memory requirements for producing the tree are relatively small, *if* subtrees that have already been evaluated are discarded (their object references are set to null). The total memory used never exceeds 15 MB.

Areas for Improvement Addressed

Of course, many of the techniques mentioned as refinements to minimax are not included in this implementation of a mancala opponent. Further research into the possibilities of parallelizing search, of compiling the code natively to a specific architecture to remove the overhead of interpretation, of rewriting the code in a language with faster execution time, of increasing hardware resources, and especially of improving the evaluation function at depth cutoff would likely prove rewarding if work on the project is extended. The evaluation function used here, while satisfactory in tests to date, returns merely the advantage in stones at the pruned node. Thus, early in the game, the computer will try for immediate gain over long-term strategy. This would become more debilitating in a game of greater depth, but it does not cause too much trouble here, as the game quickly advances such that the depth cutoff is rarely employed. Including the value of the stones still in a player's pits, weighted below the value of stones in the *mancala*, might offset this shortsightedness, as stones in pits are an asset in the endgame. Additionally, as computing power is increased, the evaluation function is used less often, and the efficiency becomes less relevant. In the absence of extensive literature on mancala strategy such as exists for chess and checkers, an opening book would prove difficult, but Schaeffer notes substantial rewards from the use of this technique in Chinook. Finally, quiescence search for extending lines of play is not recommended, as the horizon problem in mancala is particularly knotty – at any cutoff point, it is entirely possible that the next move may result in endgame, at which the balance might shift dramatically, or in capture.

Conclusion

The results of the implementation are satisfactory and useful – the program is invariably able to best its author, often by margins greater than ten stones. The implemented techniques – minimax search with alpha-beta pruning and evaluation at depth cutoff – are adequate for mancala. Stated another way, mancala is of sufficient simplicity such that these techniques are adequate – greater complexity would necessitate greater invention. Progress is still being made on refinements to these techniques, and developments in artificial intelligence can have unforeseeable ramifications in the long term. Extension of the project along the lines mentioned above could be valuable as experiential resources. Suggestions of future refinement and offers of collaboration are welcome.

Acknowledgments

The author wishes to extend thanks for the wise assistance and frequent prodding of Dr. Andy Novobilski, without whom this project would have been neither begun nor finished in any state resembling that in which it was, to Dr. Joe Dumas for inspiration and the loan of helpful literary materials, to Dr. William Lee and Dr. Jack Thompson for their participation and judgment, and to his parents.

References

- [1] Anonymous, "List of Mancala Variants," [Online document], 2003 Feb. 8, Available: http://www.wikipedia.org/wiki/List_of_Mancala_variants
- [2] J. Schaeffer, R. Lake, P. Lu, M. Bryant, "Chinook: The World Man-Machine Checkers Champion," *AI Magazine*, Volume 17, Number 1, pp. 21-29, 1996.
- [3] Anonymous, "Instructions for Mancala," [Online document], Available: http://www.pressmangames.com/instructions/instruct_mancala.html
- [4] M. D. Davis, *Game Theory: A Nontechnical Introduction*. New York: Basic Books, 1983.
- [5] J. D. Williams, *The Compleat Strategyst: being a primer on the theory of games of strategy*. New York: McGraw-Hill, 1966.
- [6] D. M. Yap, "Game Theory for Real People," *Wired*, 2002 Jul. 31, Available: <http://www.wired.com/news/print/0,1294,54131,00.html>
- [7] G. Owen, *Game Theory* (2nd ed.). New York: Academic Press, 1982.
- [8] R. D. Luce and H. Raiffa, *Games and Decisions: introduction and critical survey*. New York: Wiley, 1957.
- [9] J. von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*. Princeton: Princeton University Press, 1944.
- [10] E. Burger (Translated by J. E. Freund), *Introduction to the Theory of Games*. Englewood Cliffs, N. J.: Prentice-Hall, 1963.
- [11] N. J. Nilsson, *Principles of Artificial Intelligence*. Palo Alto, Calif.: Tioga Pub. Co., 1980.
- [12] J. Schaeffer et. al., "Chinook," [Online document], Available: <http://www.cs.ualberta.ca/~chinook/>
- [13] J. Schaeffer, "Improved Parallel Alpha-Beta Search," *Fall Joint Computer Conference*, 1986, pp. 519-527.
- [14] J. Schaeffer, *Experiments in Distributed Game-Tree Searching*. The University of Alberta, Edmonton, Alberta, Canada, January 1987.

[15] M. Newborn, "A Parallel Search Chess Program," in *1985 ACM Annual Conference Proceedings*, pp. 272-277.

[16] H. Bal and R. van Renesse, "A Summary of Parallel Alpha-Beta Search Results," *ICCA Journal*, September 1986, pp. 146-149.

[17] D. E. Knuth and R. W. Moore, "An Analysis of Alpha-Beta Pruning," in *Artificial Intelligence*, v.6, 1975, pp. 293-326.

Appendix I: Source code

```
/*
 * MancalaBoard.java
 * @author Adam Cofer
 * Created on December 1, 2002, 3:52 PM
 *
 * * Copyright 2002,2003 Adam Cofer

This file is part of Mancala

Mancala is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

*
*/

package Mancala;

public class MancalaBoard extends java.lang.Object {
    private int[] pits = new int[14];
    int turn;
        /*Players 0 and 1.
        *This is not boolean so
        *we can do math with it.
        */
    int gameState;
        /*-1:Draw
        *0: Player 0 won
        *1: Player 1 won
        *2: Playing
        */

    /* Creates a new instance of MancalaBoard */
    public MancalaBoard() throws java.lang.OutOfMemoryError {
        for(int i=0; i<pits.length; i++) pits[i] = 4;
        pits[6] = 0;
        pits[13] = 0;
        gameState = 2;
    }

    /* Copy constructor */
    public MancalaBoard(MancalaBoard Board) throws java.lang.OutOfMemoryError {
        for(int i=0; i<pits.length; i++) pits[i] = Board.getPitValue(i);
        turn = Board.whoseTurn();
        gameState = Board.getGameState();
    }

    public int getPitValue(int i) {
        return pits[i];
    }

    public int whoseTurn() {
        return turn;
    }

    public int getGameState() {
        return gameState;
    }
}
```

```

}

public void setGameState() {
    if ( isEndGame() ) {
        /* collect stones first */
        for( int i=0; i<=5; i++ ) {
            pits[6] += pits[i];
            pits[i] = 0;
        }
        for( int i=7; i<=12; i++ ) {
            pits[13] += pits[i];
            pits[i] = 0;
        }
        if ( pits[6] > pits[13] ) gameState = 0;
        else if ( pits[13] > pits[6] ) gameState = 1;
        else if ( pits[13] == pits [6] ) gameState = 3;
        else gameState = -1;
    } else gameState = 2;
}

public boolean isLegalMove(int i) {
    if(i == 6 || i == 13) {
        return false;
    }
    if( i > 13 ) {
        return false;
    }
    if( turn == 0 || turn == 1) {
        if( (pits[i] > 0) && ( 7*turn) <= i ) && ( i <= 5+(7*turn) ) ) {
            return true;
        } else {
            return false;
        }
    } else {
        System.out.println("isLegalMove() Error: cannot tell whose turn it is.");
        return false;
    }
}

public boolean isEndGame() {
    boolean temp = true;    //Assume it's the end
    int i = 0;
    for( i=0; i <= 5; i++) if ( pits[i] > 0 ) temp = false;
    if( false == temp ) {
        temp = true;
        for( i=7; i <=12; i++) if ( pits[i] > 0 ) temp = false;
    }
    return temp;
}

public void move(int i) {
    if( !isLegalMove(i) ) {
        System.out.println("Warning! " + i + " is not a legal move.");
        return;
    }
    if ( gameState() != 2 ) {
        System.out.println( "move() Error: trying to play after game is over." );
        return;
    } else {
        int temp = pits[i];    //"pick up" the stones
        pits[i] = 0;          // ""
        while(temp > 0) {     //deposit the stones
            i++;
            if( i >= 14 ) i = i-14;    /*keep array index in bounds,
                                     *while 'wrapping around'

```


You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
*/

```
package Mancala;

public class MancalaNode extends java.lang.Object {
    public MancalaBoard thisBoard;
    private int depth;
    private int value; /* this is where we'll store the minimax value */
    private boolean visited;
    private MancalaNode parent;
    private MancalaNode children[] = new MancalaNode[14];

    /* empty constructor, I don't expect this to be used, but it can't hurt. */
    public MancalaNode() throws java.lang.OutOfMemoryError {
        thisBoard = new MancalaBoard();
        depth = 0;
        visited = false;
        parent = null;
        for (int i = 0; i <= 13; i++) children[i] = null;
        value = -999;
    }

    /* the constructor that I'll actually be using, where
    * we pass a Board in for the Node to hold and a depth */
    public MancalaNode(MancalaBoard nodeBoard, int nodeDepth) throws java.lang.OutOfMemoryError {
        thisBoard = nodeBoard;
        depth = nodeDepth;
        visited = false;
        /* we'll set parents, children, to useful values with methods later */
        parent = null;
        for (int i = 0; i <= 13; i++) children[i] = null;
        value = -999;
    }

    public MancalaBoard getBoard() {
        return thisBoard;
    }

    public int getDepth() {
        return depth;
    }

    public void setDepth(int newDepth) {
        depth = newDepth;
    }

    public int getValue() {
        return value;
    }

    public void setValue(int newValue) {
        value = newValue;
    }

    public boolean getVisited() {
        return visited;
    }

    public void setVisited(boolean newVisited) {
        visited = newVisited;
    }

    /* setting parents, children, and siblings */
    public MancalaNode getParent() {
```

```

    return parent;
}

public void setParent(MancalaNode parentNode) {
    parent = parentNode;
}

/* a little more complicated to get and set the children */
public MancalaNode getChild(int whichChild) {
    if ( (whichChild < 0) || (whichChild > 13) ) {
        System.out.println("MancalaNode.getChild(): prevented ArrayIndexOutOfBoundsException.");
        return this;
    }
    return children[whichChild];
}

public void setChild(MancalaNode childNode, int whichChild) {
    if ( (whichChild < 0) || (whichChild > 13) ) {
        System.out.println("MancalaNode.setChild(): prevented ArrayIndexOutOfBoundsException.");
        return;
    }
    children[whichChild] = childNode;
}
}

/*
 * MancalaServer.java
 * Author: Adam Cofer
 * Department of Computer Science, University of Tennessee at Chattanooga
 *
 * A computer opponent and interface for Mancala
 *
 * Copyright 2002,2003 Adam Cofer

This file is part of Mancala

Mancala is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
*
*
* Developed using the Sun JDK 1.4.1, www.sun.com
* This program is best run on the Sun JRE 1.4.1,
* from the command line enter "java Mancala/MancalaServer"
* or, if you received this program as a .jar file, enter
* "java -jar Mancala.jar"
*/

package Mancala;

import java.io.*;
import java.util.*;

public class MancalaServer extends java.lang.Object {

```

```

/* MAXIMUM is used to set the depth of our search, see findBestMove() */
public static final int MAXIMUM = java.lang.Integer.MAX_VALUE;
public static final int DEPTH_CUTOFF = 11;
private static MancalaBoard BoardInPlay = new MancalaBoard();
private static MancalaUI UIInPlay = new MancalaUI();
private static int maxDepthFound = 0;
private static long nodesCreated = 0;
private static long[] nodesAtEachDepth = new long[100];
private static Date start;
private static long startLong;
private static long cutoffs = 0;

/* Creates a new instance of MancalaServer */
public MancalaServer() {
}

public static void main(String[] args) {
    for( int i=0; i<=99; i++ ) nodesAtEachDepth[i] = 0;

    /* this is the main program loop */
    while( !(BoardInPlay.isEndGame()) ) {
        if (BoardInPlay.whoseTurn() == 0){
            UIInPlay.draw(BoardInPlay);
            int i = UIInPlay.requestMove(BoardInPlay);
            if (i == -1) break;
            if ( (i<0) || (i>12) ){
                System.out.println("");
            } else BoardInPlay.move(i);
        } else if (BoardInPlay.whoseTurn() == 1){
            MancalaNode evalNode = new MancalaNode(BoardInPlay, 0);
            nodesAtEachDepth[0] = 1;
            for(int q=1; q <= 99; q++) nodesAtEachDepth[q] = 0;
            nodesCreated = 0;
            cutoffs = 0;
            start = new Date();
            startLong = start.getTime();
            System.out.println("Evaluating moves...");
            int i = findBestMove(evalNode);
            System.out.println("");

            System.out.println("Moving from pit " + i + ".");
            BoardInPlay.move(i);
        } else {
            System.out.println("main() error: turn is indeterminate.");
            return;
        }
    }

    BoardInPlay.setGameState(); /* Finish up by determining who won*/
    System.out.println("Game over, man. Game over.");
    UIInPlay.draw(BoardInPlay);
    switch(BoardInPlay.getGameState()) {
        case 0:
            System.out.println("Player won!");
            break;
        case 1:
            System.out.println("Computer won!");
            break;
        case 2:
            System.out.println("Quitter.");
            break;
        case 3:
            System.out.println("We have a tie!");
            break;
        case -1:
    }
}

```

```

        System.out.println("This should never happen.");
        break;
    default:
        System.out.println("main() Error: value returned from getGameState() seriously weird.");
        break;
    }
}

private static int findBestMove(MancalaNode currentNode){
    int bestMoveFound = -1;

    int turn = currentNode.thisBoard.whoseTurn();
    int offset = turn * 7;
    /* value should be -999 if turn is 1 (i.e. computer's/MAX/alpha)
    * 999 if turn is 0 (i.e. player's/MIN/beta) */
    if( 1 == turn ) { currentNode.setValue(-999);
    } else if( 0 == turn ) { currentNode.setValue(999);
    } else System.out.println("error in findBestMove(): turn indeterminate.");

    for( int i=(0+offset); i<=(5+offset); i++ ) {
        if( currentNode.thisBoard.isLegalMove(i) ) {
            if((currentNode.getDepth()==0)&&(currentNode.thisBoard.whoseTurn()==1)) {
                System.out.print(".");
            }
            try {
                MancalaBoard newBoard = new MancalaBoard(currentNode.thisBoard);
                newBoard.move(i);
                /* create a new node with this board and set its depth to current depth + 1 */
                MancalaNode newNode = new MancalaNode(newBoard, currentNode.getDepth()+1);

                nodesCreated++;
                if( newNode.getDepth() <= 99 ) nodesAtEachDepth[newNode.getDepth()]++;
                if(0 == nodesCreated % 1000000) {
                    System.out.print(".");
                }

                newNode.setParent(currentNode);
                currentNode.setChild(newNode, i);
                /* check for endgame, if true set a value in the node*/
                if( newNode.thisBoard.isEndGame() ) {
                    newNode.thisBoard.setGameState();
                    newNode.setValue(newNode.thisBoard.getPitValue(13) - newNode.thisBoard.getPitValue(6));
                } else if( newNode.getDepth() > DEPTH_CUTOFF ) {
                    /* Depth cutoff evaluation function */
                    newNode.setValue(newNode.thisBoard.getPitValue(13) - newNode.thisBoard.getPitValue(6));
                } else { /* if not endgame, we need to expand the node */
                    findBestMove(newNode);
                }
            }
            if(currentNode.thisBoard.whoseTurn() == 1 ) { /* 1/computer/MAX/alpha can never decrease */
                if( currentNode.getChild(i) != null ) {
                    if( currentNode.getChild(i).getValue() > currentNode.getValue() ) {
                        currentNode.setValue(currentNode.getChild(i).getValue());
                        bestMoveFound = i;
                    }
                }
            }
            currentNode.setChild(null, i); /* memory management */
        }
    }
    /* ALPHA CUTOFF */
    /* cutoff if 1/computer/MAX/alpha value is greater than ANY
    * 0/player/MIN/beta parent value */
    MancalaNode nodePtr = currentNode;
    while( nodePtr.getParent() != null ) {
        nodePtr = nodePtr.getParent();
        if((nodePtr.thisBoard.whoseTurn()==0) && (currentNode.getValue() > nodePtr.getValue())){
            nodePtr = null;
            cutoffs++;
        }
    }
}

```

```

        return bestMoveFound;
    }
}
nodePtr = null; /* cleanup both here and above to account for cutoff/no cutoff cases */
/* END ALPHA CUTOFF */
}
if(currentNode.thisBoard.whoseTurn() == 0 ) { /* 0/player/MIN/beta can never increase */
    if( currentNode.getChild(i) != null ) {
        if( currentNode.getChild(i).getValue() < currentNode.getValue() ) {
            currentNode.setValue(currentNode.getChild(i).getValue());
            bestMoveFound = i;
        }
    }
    currentNode.setChild(null, i);
/* BETA CUTOFF */
    /* cutoff if 0/player/MIN/beta value is less than ANY
    * 1/computer/MAX/alpha parent value */
    MancalaNode nodePtr = currentNode;
    while( nodePtr.getParent() != null ) {
        nodePtr = nodePtr.getParent();
        if((nodePtr.thisBoard.whoseTurn()==1) && (currentNode.getValue() < nodePtr.getValue())){
            nodePtr = null;
            cutoffs++;
            return bestMoveFound;
        }
    }
    nodePtr = null;
/* END BETA CUTOFF */
}
} catch (java.lang.OutOfMemoryError e) {
    System.out.print(",");
    return -1;
}
}
}
return bestMoveFound;
}
}
/*
* MancalaUI.java
*
* Created on December 2, 2002, 7:02 PM
* Modified February 6, 2003
* Modified March 17, 2003 (reversed board presentation in draw())
* Copyright 2002,2003 Adam Cofer

This file is part of Mancala

Mancala is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
*/

```

```
package Mancala;
```

```
import java.io.*;
```

```

public class MancalaUI extends java.lang.Object {

    private StringBuffer[] buffer = new StringBuffer[14];
    private char[][] pitHolder = new char[14][4];

    /* Creates a new instance of MancalaUI */
    public MancalaUI() {
        for ( int i=0; i < buffer.length; i++) {
            buffer[i] = new StringBuffer("");
            pitHolder[i] = "   ".toCharArray();
        }
    }

    public void draw(MancalaBoard Board) {

        for ( int j=0; j < buffer.length; j++) {
            buffer[j].delete(0, buffer[j].length());
            pitHolder[j] = "   ".toCharArray();
            buffer[j].append(Board.getPitValue(j));
            if(buffer[j].length() == 1) {
                pitHolder[j][3] = buffer[j].charAt(0);
            } else if(buffer[j].length() >= 2) {
                pitHolder[j][3] = buffer[j].charAt(1);
                pitHolder[j][2] = buffer[j].charAt(0);
            } else System.out.println("draw() error: empty StringBuffer");
        }

        int i = 0;

        System.out.println("");

        System.out.println("  [12][11][10][ 9][ 8][ 7]");

        System.out.print("   ");
        for ( i = 12; i >= 7 ; i--) System.out.print( pitHolder[i] );
        System.out.println("");

        System.out.print(pitHolder[13]);
        System.out.print(" ----- ");
        System.out.print(pitHolder[6]);
        System.out.println();

        System.out.print("   ");
        for ( i = 0; i <= 5 ; i++) System.out.print( pitHolder[i] );
        System.out.println("");

        System.out.println("  [ 0][ 1][ 2][ 3][ 4][ 5]");
    }

    public int requestMove(MancalaBoard Board){
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line = " ";

        System.out.println("Player " + Board.whoseTurn() + " enter a move or type 'quit' to exit:");
        try {
            line = in.readLine();
            if (line.equals("") || line.equals("quit"))
                return -1;
            StringBuffer buf = new StringBuffer(line);
            String buff = new String(buf);
            int i = java.lang.Integer.parseInt(buff);
            if( (i < 0) || (i > 12) ){
                System.out.println(i + " is not a valid move. Please enter a number in the range [0,5]");
                return -4;
            }
        }
    }
}

```

```
        return i;
    }
    catch (java.io.IOException e) {
        System.out.println("You must enter something, preferably a number.");
        return -2;
    }
    catch (java.lang.NumberFormatException e) {
        System.out.println("Please enter a valid number.");
        return -3;
    }
}
```