Lines of Action - Artificial Intelligence

Mario Garcia
Johns Hopkins University
Engineering for Professionals

***Abstract*—The problem of simulating intelligence, known as Artificial Intelligence (AI), has been at the forefront of modern technology with computer scientists and software engineers having created many algorithms that computers use to make decisions. A subset of these algorithms pertains to the intelligence of playing board games. Search trees and heuristics are extremely useful in analyzing all the possible moves available to the computer and deciding which is the most advantageous. This article deals with the game Lines of Action (LOA) and details how a search tree (also known as a game tree) and a heuristic evaluation can be used in creating an algorithm that plays LOA intelligently. This algorithm for playing LOA is comparable to algorithms for playing Chess, Othello, and many other "perfect information" games. The problem of creating AI for these types of board games with high state-space complexity is often solved with the use of search trees and heuristics, some of which are extremely sophisticated.**

I. Introduction

Lines of Action (LOA) is a two-player connection game in which each player attempts to connect all of his/her pieces. It is played on a chess board (8x8 grid) and each player starts with 12 pieces. This leads to an enormous amount of possible game states. As such, solving the problem of creating an algorithm to play LOA contributes to the general field of Artificial Intelligence by offering a methodology to approaching problems with high state-space complexity.

There are several types of search trees and various heuristics that can be used to create such an algorithm. The type of search tree used will impact the run time and memory usage of the algorithm, while the chosen heuristic will greatly impact the effectiveness with which the algorithm selects the best moves.

This paper will present an algorithm that efficiently plays LOA. The algorithm uses a minmax depth-limited search tree with $\alpha\beta$ pruning and implements a combination of a quad heuristic and a centralization heuristic as described by Winands [2] to evaluate game states.

It is unlikely that the algorithm presented in this paper will be able to seriously compete with the algorithms employed by MIA (Maastricht in Action), the computer program developed by Mark Winands from which this algorithm derives its heuristic, or the computer programs YL and Mona developed at the University of Alberta by Yngvi Bjornsson and Darse Billings respectively [3]. Nonetheless, this algorithm serves as a basis for future research and modifications to include more sophisticated search and evaluation techniques and allows for the exploration into the usefulness of the techniques described herein.

The remainder of this paper is structured as follows. In Section II background information is provided including a brief history of LOA, an explanation of the game, and some relevant work by other computer scientists in reference to LOA. Section III offers an in-depth description of the methodology and core components of the presented algorithm. Findings and analysis are provided in Section IV. And lastly, the conclusion is presented in Section V.

II. Literature Review

This section provides the background information necessary to understand the problem of creating an algorithm to play LOA intelligently. It offers a history of Lines of Action including information about the game and how it's played. It then presents a brief discussion of relevant work before closing with a comparison between the presented algorithm and an algorithm that selects moves at random.

A. History

Lines of Action was created by Claude Soucie and first publicized in 1969 by Sid Sackson in his book *A Gamut of Games* [5]. However, not until the 1990s did its popularity begin to rise. In 1997 the first Mind Sports Olympiad was held by the Mind Sports Organization in London's Royal Festival Hall which included the Lines of Action world championship and continues to host the event annually [7], [8]. In 2000 Lines of Action was included in the 5th Computer Olympiad with 3 computer contestants [6]. Lines of Action also continues to be included in this annual event.

B. The Game

Lines of Action is a a zero-sum game (meaning that the gains of one player are exactly balanced by the losses of the other) with perfect information (meaning that no information about the game state is ever hidden from either player). The game is played on an 8x8 grid (such as a standard chessboard). The game starts with 24 pieces, 12 black and 12 white. One player controls the black pieces and the other player controls the white. The rules are as follows.

1. The black pieces are set up with 6 at the top of the board and 6 at the bottom, while the white pieces are set up with 6 at the left side of the board, and 6 the right. The initial setup arrangement is shown in Fig.1.
2. Players alternate turns with black having the first move of the game.
3. Pieces can move vertically, horizontally, or diagonally.
4. The total number of pieces on a given line is the number of spaces a piece can move along that line (these are the *Lines of Action*). Some possible moves from different board positions are shown in Fig.2.

5. A piece can jump over pieces of the same color.
6. A piece can not jump over opponent pieces but can capture them by landing on them.
7. A player wins by connecting all of his/her pieces vertically, horizontally, and/or diagonally into one contiguous unit as shown in Fig.3.
8. If a move simultaneously creates a connected unit for both players, the player having moved wins.
9. If a player can not move, this player loses.

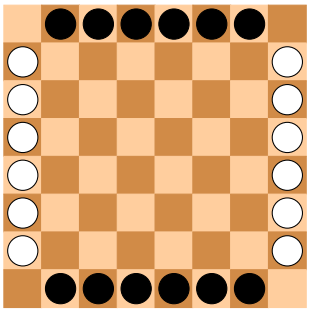
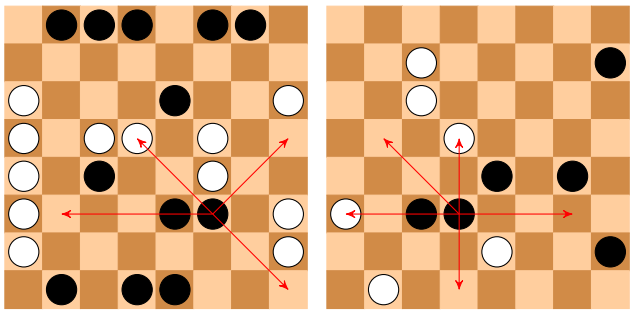


Fig. 1: Starting Position



(a) (b)

Fig. 2: Possible Moves

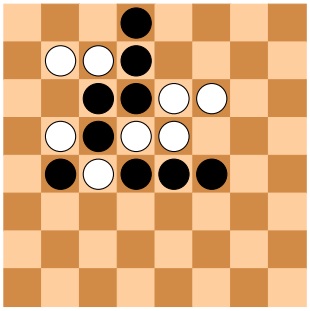


Fig. 3: Win Condition (Black Wins)

C. Relevant Work

One of the most well-known computer scientists to study Lines of Action is Mark Winands. In August 2000 Winands submitted his M.S. Thesis on the topic [1]. A great deal of the information in this paper can be attributed to Winands's thesis. In his thesis he details the inner workings of his LOA program MIA (Maastricht in Action). He outlines the important components to evaluating the game states of LOA along with three evaluation functions, each addressing some unique intricacy of LOA. His quad evaluator, which focuses on the solidness and connectedness of the formations, is implemented in the algorithm described in the following section.

D. Comparison

As previously stated the algorithm described in this paper uses a depth-limited search tree to find a best move. Essentially this means the AI does not only look at the benefits of a current move but also looks ahead a set number of moves to see how the current move may affect future moves. A more simplistic algorithm by which a computer could play LOA would be one that only considers the benefits of the current move and does not factor the impact it may have later in the game. Although these two algorithms are fundamentally different, they can be implemented quite similarly (in fact, the second algorithm can be implemented as the first with a depth of 1). This makes the second algorithm an excellent mechanism against which the algorithm in this paper can be benchmarked. This comparison is performed in the *Findings and Analysis* section (Section IV) and shows the effectiveness of using a search tree.

III. Methodology

This section presents the algorithm. This algorithm will play LOA intelligently. As previously stated, it may not always win, but it will solve the problem of having a computer play LOA intelligently and will provide a foundation from which to build upon.

A. Problem

Due to the number of starting pieces and the size of the board, LOA has approximately 1×10^{24} possible game states [1]. This high level of state-space complexity is essentially the main problem in creating AI for Lines of Action. The enormous amount of possible game states makes using a database, of all positions and the best moves to make from those positions, very impractical and sometimes nearly impossible.

B. Approach

1) MinMax Search Tree: One of the most common approaches that addresses the problem of high state-space complexity in games is to use search trees, also known as game trees. Using a search tree is efficient because the algorithm does not need to store every possible state and later search for the best move, but can instead search for the best move while creating the game tree. Every possible game state is represented as a node in the search tree. In creating the search tree the algorithm is essentially simulating a game. The root node of the game tree is the current position. Each child is the position resulting from a possible move being made by the other player. Each level of the tree is referred to as a ply and each ply simulates the moves made by one of the two players.

The algorithm selects the best move by analyzing the resultant states of each move and selecting the one that maximizes

the computer's advantage. However, the algorithm assumes that its opponent will maximize his/her own advantage as well, in turn minimizing the computer's (based on the principal of zero-sum). As such, the algorithm selects moves that offer its opponent the least possible advantage (even when making his/her best possible move) while simultaneously giving itself the greatest advantage. This is known as a "minmax" algorithm which recursively searches the game tree.

The pseudo code for the minmax algorithm[1] is presented below. The procedures MINVALUE and MAXVALUE receive a node n of the game tree as input and use a VALUE procedure (which is described later) to evaluate the advantageousness of a given position. MINVALUE calls a MINIMUM procedure that returns the minimum value between two received values. Similarly, the MAXVALUE calls a MAXIMUM procedure that returns the maximum of two received values. Keep in mind that the term "leaf node" in the MAXVALUE and MINVALUE may not necessarily be a terminal state (where the game is over) but can be a leaf node due to a set number of plies to which the game tree is created. This can be thought of as the number of moves the computer thinks ahead (this concept is discussed in detail later in the paper).

```
MINMAX()
1  n = current game state
2  max = MAXVALUE(n)
3  make move associated with max

MAXVALUE(n)
1  if n is a leaf node
2       return VALUE(n)
3  max = −∞
4  for each child of n
5       max = MAXIMUM(max, MINVALUE(child))
6  return max

MINVALUE(n)
1  if n is a leaf node
2       return VALUE(n)
3  min = ∞
4  for each child of n
5       min = MINIMUM(MAXVALUE(child))
6  return min
```

2) $\alpha\beta$ Search Tree: The minmax algorithm is made more efficient by using a method known as $\alpha\beta$ pruning. This method increases the speed of the search by decreasing the number of nodes in the game tree. In creating the game tree the algorithm remembers the result of a simulated move. If it simulates a move that will lead to a worse position, it stops the simulation of the game and moves on with its search. It does not care exactly how much worse the position will be since it knows it will not select the move that leads to it. It applies this same logic to the opponents simulated moves. In this way it can prune entire branches of the game tree that need not be searched.

The algorithm for the $\alpha\beta$ search tree is presented as follows. As in the minmax algorithm, the ALPHAVALUE and BETAVALUE receive a node n of the game tree as an input, use a VALUE procedure to evaluate a given positions, and use MAXIMUM and MINIMUM procedures respectively. As additional inputs, these procedures receive $alpha$ and $beta$ values that identify the maximum and minimum values of previously searched nodes. These values allow the algorithm to know which nodes can be pruned.

```
ALPHABETA()
1  n = current game state
2  alpha = ALPHAVALUE(n, −∞, ∞)
3  make move associated with alpha

ALPHAVALUE(n, alpha, beta)
1  if n is a leaf node
2       return VALUE(n)
3  v = −∞
4  for each child of n
5       v = MAXIMUM(v, BETAVALUE(child, alpha, beta))
6       if v ≥ beta
7            return v
8       alpha = MAXIMUM(v, alpha)
9  return v

BETAVALUE(n, alpha, beta)
1  if n is a leaf node
2       return VALUE(n)
3  v = ∞
4  for each child of n
5       v = MINIMUM(v, ALPHAVALUE(child, alpha, beta))
6       if v ≤ alpha
7            return v
8       beta = MINIMUM(v, beta)
9  return v
```

3) Quad Heuristic: Although the search tree addresses the problem of high state-space comlexity, it does not entirely solve it. An underlying problem exits in how the algorithm evaluates the advantageousness of a position. Naturally positions that lead to a win for the computer are advantageous and positions that lead to a loss are not. For a game like Tic-Tac-Toe which has approximately 2×10^4 possible game states[2] the entire game tree can be created, in turn, simulating every possible game to its conclusion, known as a terminal state (win, lose, or draw). This makes selecting the best move fairly straightforward since the possible end results of each move are known. The algorithm simply avoids moves that lead to a loss and makes moves that lead to a win or a draw.

However, as previously stated, LOA has about 1×10^{24} possible game states. An algorithm that creates an entire game tree for LOA would be more practical than the use of a database but would, nonetheless, still be impractical. If the

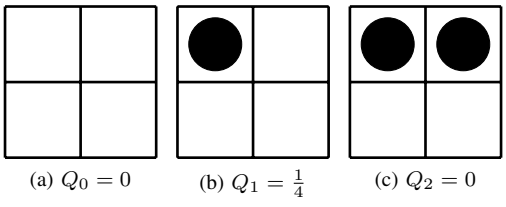
[1]The pseudo code for the minmax algorithm is adapted from [4].

[2]Three possibilities (X, O, or blank) for each of the nine spaces on the board resulting in $3^9 = 19,683$ possible game states.

entire tree is not created, the end result of each simulated game will not be known, and the algorithm will have to decide the advantageousness of each position in some other way.

To solve this problem an evaluation function is created to evaluate non-terminal states. This evaluation function and the level of advantageousness it assigns to non-terminal game states is known as a heuristic. Each move is assigned a heuristic value based on the evaluation of the resulting game state. Heuristics are effectively the "brains" behind the decision making process of AI in games because they add a weight factor to each possible move.

The algorithm being presented here utilizes two separate heuristic methods. These heuristic methods make up the VALUE procedure in the MINMAX and ALPHABETA procedures. The first is a quad heuristic as described by Mark Winands [2]. This heuristic generally gives the number of connected units for one player in the following way. It breaks up the board into 81 2x2 grids called quads. It then assigns a weighted value to each of these quads based on the number of pieces in the quad[3] and their arrangement[4] within the quad as shown in Fig.4.



(a) $Q_0 = 0$ (b) $Q_1 = \frac{1}{4}$ (c) $Q_2 = 0$

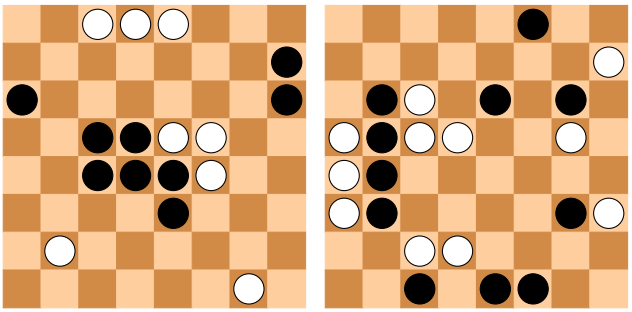
(d) $Q_3 = -\frac{1}{4}$ (e) $Q_4 = 0$ (f) $Q_d = -\frac{1}{2}$

Fig. 4: Quad Types and Euler Contributions

Lastly, it uses the weighted values of each quad to calculate a Euler number which represents the advantageousness of the position. Some example game states with their corresponding Euler numbers are shown in Fig.5. Greater Euler numbers mean that the player has a greater number of unconnected units making the position less advantageous for that player. Because smaller numbers maximize the advantage of a position, the Euler number needs to be negative for the minmax search. For example, a Euler number of 3 indicates a better position for a given player than a Euler number of 5. However, because the minmax algorithm selects the greatest of all the options it would select the position with a Euler number of 5 which would not be the best choice. If instead -3 and -5 are returned as the corresponding value of these positions the minmax algorithm would select the position with a value of -3 which would be the best choice according to the heuristic.

[3]Any part of a quad that lies outside of the board is considered empty.

[4]There are essentially only 6 quad arrangements due to rotational equivalence.



(a) $White = 4$; $Black = 3$ (b) $White = 6$; $Black = 7$

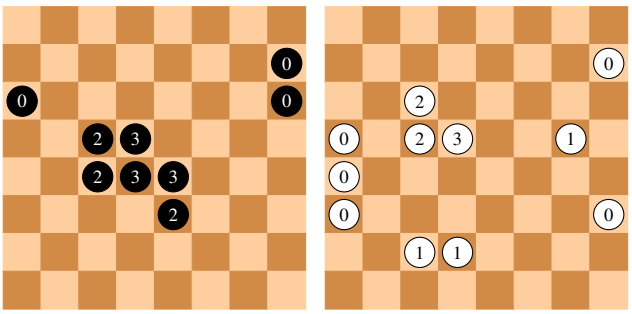
Fig. 5: Euler Values

4) Centralization Heuristic: The second heuristic is much simpler. It assigns a greater value to pieces closer to the center of the board. The logic is that the center of the board is the area to which all pieces can converge most easily because the center has the smallest average distance to all pieces. The centralization heuristic assigns values to each location on the board as shown in Fig.6.

0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0
0	1	2	2	2	2	1	0
0	1	2	3	3	2	1	0
0	1	2	3	3	2	1	0
0	1	2	2	2	2	1	0
0	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0

Fig. 6: Weighted Board Values

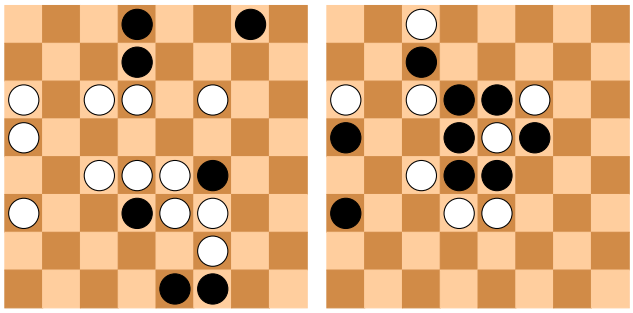
Using this weighted board, a value can be calculated that determines how close the pieces are to the center. This is done by summing up the values corresponding to the locations at which the player has pieces. Some examples of positions with their corresponding centralization values can be seen in Fig.7.



(a) $Value = 15$ (b) $Value = 10$

Fig. 7: Centralization Values

The final heuristic algorithm considers both the computer's and the player's advantage given a particular move. This means that it calculates a heuristic value (using both quad and centralization heuristic methods) for the computer's pieces and the subtracts from it the heuristic value of the player's pieces. In this way, the AI realistically attempts to thwart its opponent. Example game states with the final heuristic values for the AI's position (playing as white) are show in Fig.8. These are the return values of the VALUE procedure previously mentioned in the algorithms for minmax and alpha-beta search. The individual contributions of the quad and centralization heuristics for the figures in Fig.8a and Fig.8b are shown in TABLES I and II respectively.



(a) heuristic value $= 14$; (b) heuristic value $= -5$;

Fig. 8: Final Heuristic Values with Respect to White

TABLE I: Individual Contributions to Final Value in Fig.8a

	Black	White	Total
Quad	5	-5	0
Centralization	-5	19	14
			14

TABLE II: Individual Contributions to Final Value in Fig.8b

	Black	White	Total
Quad	3	-5	-2
Centralization	-16	13	-3
			-5

5) Results: With the inclusion of the quad and centralization heuristics the game tree can now be limited to a certain depth without the need for creating the entire game tree with terminal states. The depth will be limited to a fixed value. The resultant tree is called a depth-limited search tree. The algorithm will create the search tree to some predetermined depth at which point it will evaluate the nodes in the last ply as if they were terminal states. Terminal states will still be checked for in which case the actual outcome of the game with that position will supersede its heuristic value.

It turns out that checking for terminal states can become rather costly in terms of time. The use of the quad heuristic adds a way to decrease the time spent on checking for terminal states. The Euler number of the quad heuristic is essentially the number of connected units a player has. As such, if the number is greater than 1 the position can not be a terminal state and the terminal check can be skipped. The quad heuristic offers an increase in speed primarily because not all 81 quads need to be recalculated after every move. Moves that result in a capture require that twelve quads be updated, and moves that do not result in a capture require that only eight quads be updated.

However, it is possible for non-terminal states to have a Euler number of 1. This means that the quad heuristic can only verify non-terminal states and does not identify terminal states. Nonetheless, since a vast majority of the positions evaluated will be non-terminal the quad heuristic saves the algorithm some time in unnecessarily checking these positions for a terminal condition. This is

C. Architecture

The architecture of this system is a simple graphical user interface (GUI). It allows the user to select a depth level for which the depth-limited search tree will be created which essentially allows the user to vary the difficulty of playing against the AI. It also shows the user all of the possible moves for a selected piece. Lastly, it logs the moves played throughout the game in the event that the user wishes to analyze a given game.

D. Realization

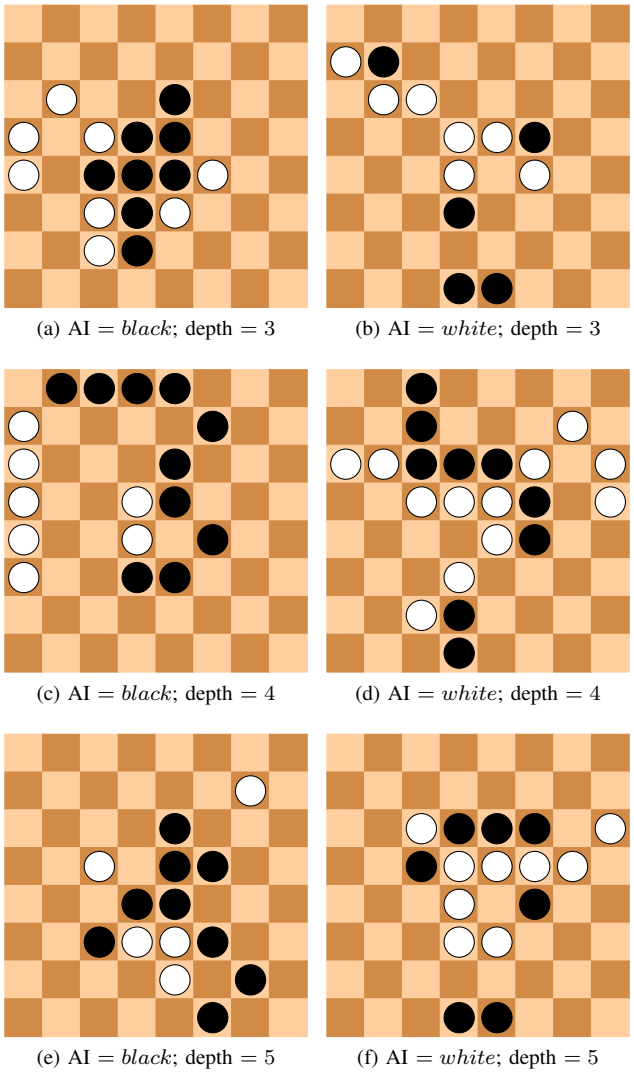
The algorithm in this paper is implemented as part of Dr. Gilbert Peterson's Lines of Action Java Applet. The applet was previously a fully functional applet devoid of artificial intelligence. It had all the necessary GUI functionality and pertinent data structures in place to facilitate the adaption of the algorithm presented herein to the applet, offering it the missing AI. The applet was developed in the Eclipse IDE (Integrated Development Environment) using the JDK (Java Development Kit) Version 1.6.

IV. FINDINGS AND ANALYSIS

A. Effectiveness of Search Tree

As should be apparent by now, a search tree allows the AI to look ahead and weigh a current move against future states of the board based on that move. An opposing algorithm would be one that does not look ahead at all but only analyzes the benefits of the current move being made. The number of moves the AI looks ahead is the depth of the search tree. As such, the second algorithm is essentially a subset of the first in which the AI is only looking ahead one move (the current move being made). In order to test the effectiveness of using a search tree, the two algorithms were compared by having them play eight games against each other.

The first algorithm (the algorithm presented in this paper) will be referred to simply as "AI" for the remainder of this section. In the first two games the depth for "AI" was set to three, in the third and fourth games it was set to four, and in the last two games it was set to five. At each depth "AI" played as black and then as white. "AI" won every game. The final board positions are shown in Fig.9.



(a) AI = *black*; depth = 3 (b) AI = *white*; depth = 3

(c) AI = *black*; depth = 4 (d) AI = *white*; depth = 4

(e) AI = *black*; depth = 5 (f) AI = *white*; depth = 5

Fig. 9: Final Board Positions of Comparative Test

In this test, the method in which the best move is selected was isolated from other variables primarily because both algorithms used the same heuristic evaluation method. The results of this test indicate that using a search tree is indeed a more effective method of selecting advantageous moves than not using one. Although further testing would have to be conducted in order to quantize the level of added effectiveness, these tests serve as a benchmark and confirm that there is an added benefit to using a search tree.

B. Effectiveness of $\alpha\beta$ pruning

As previously stated, using an $\alpha\beta$ pruning technique can increase the efficiency of the minmax search algorithm. In order to verify this claim a test was conducted in which eight games (two sets of four games each) were played against the AI. The AI used the minmax search algorithm in one set of games and the $\alpha\beta$ search algorithm in the other. The sequence of moves made against the AI were identical in the two sets of games so that the AI would create the exact same search tree in both cases and would only alter the way in which it searched the game tree. TABLE III shows the resultant data from this test. The "% $\searrow$" columns show the decrease (as percentages) of either the nodes searched or the time taken from using $\alpha\beta$ pruning.

TABLE III: Minmax vs. $\alpha\beta$

		Nodes Searched			Time (s)		
Game #	Tree Nodes	Min Max	$\alpha\beta$	% $\searrow$	Min Max	$\alpha\beta$	% $\searrow$
1	44631	44631	9150	79.50	0.052	0.012	76.92
	50130	50130	10716	78.62	0.053	0.016	69.81
	58199	58199	11322	80.55	0.064	0.016	75.00
	69312	69312	13801	80.09	0.075	0.019	74.67
	65785	65785	12646	80.78	0.074	0.018	75.68
	64704	64704	13769	78.72	0.07	0.02	71.43
	60809	60809	10559	82.64	0.068	0.016	76.47
	42592	42592	8970	78.94	0.046	0.012	73.91
	56184	56184	10793	80.79	0.061	0.016	73.77
	57284	57284	10135	82.31	0.061	0.015	75.41
	70532	69621	14721	78.86	0.075	0.02	73.33
2	42861	42861	12864	69.99	0.047	0.018	61.70
	59831	59831	14445	75.86	0.068	0.018	73.53
	46388	46388	14422	68.91	0.052	0.02	61.54
	67173	67173	16361	75.64	0.077	0.021	72.73
	60316	60316	17904	70.32	0.07	0.022	68.57
	55149	55149	19643	64.38	0.064	0.025	60.94
	62385	62385	17323	72.23	0.074	0.025	66.22
	48530	48530	16393	66.22	0.054	0.022	59.26
	48424	48424	9954	79.44	0.052	0.014	73.08
	38082	37956	5482	85.56	0.043	0.008	81.40
3	61438	61438	12614	79.47	0.066	0.023	65.15
	57094	57094	10106	82.30	0.061	0.016	73.77
	78651	78651	14351	81.75	0.086	0.021	75.58
	57936	57936	11326	80.45	0.064	0.016	75.00
	48424	48424	10123	79.10	0.053	0.014	73.58
	42592	42592	8552	79.92	0.045	0.011	75.56
	57445	57445	11118	80.65	0.062	0.016	74.19
	58932	58932	13542	77.02	0.07	0.018	74.29
	64626	62541	14205	77.29	0.071	0.02	71.83
4	62814	62814	18644	70.32	0.07	0.024	65.71
	57933	57933	14094	75.67	0.067	0.019	71.64
	59417	59417	11968	79.86	0.068	0.017	75.00
	53009	53009	13709	74.14	0.059	0.023	61.02
	59946	59946	15807	73.63	0.068	0.021	69.12
	54303	54303	18883	65.23	0.064	0.025	60.94
	51925	51925	16995	67.27	0.061	0.023	62.30
	44631	44631	9309	79.14	0.048	0.013	72.92
	50130	50130	10736	78.58	0.054	0.013	75.93
	58199	56437	11011	80.49	0.062	0.016	74.19

This test shows that using an $\alpha\beta$ search technique prunes 77% of the game tree on average. This results in an average increase in speed of about 71% which is extremely noticeable. This means that if the AI is given a limited amount of time to search the tree and make a move, the $\alpha\beta$ search technique will utilize this time more efficiently resulting in overall better play. This is apparent in the implemented applet when the depth is set to 5 plies and the time is set to 5 seconds. When using the minmax search algorithm the AI always runs out of time and does not get to search the entire tree, meaning that there may have been a better move available than the one it chose but it simply ran out of time prior to having evaluated that move in the tree. On the other hand, the $\alpha\beta$ search finishes its entire search of the tree in the allotted time and so is guaranteed to have selected the best possible move within the tree of 5 plies.

V. Conclusion

The methods herein proposed serve as a solution to the problem of artificial intelligence for Lines of Action. The

usefulness of a minmax search tree as an effective means of artificial intelligence has been verified by comparing it to an artificial intelligence that lacks the use of a search tree. This usefulness is further improved by way of an $\alpha\beta$ pruning technique which the test data verifies. The importance of heuristics at analyzing those search trees has been clearly demonstrated in the proposed algorithm. The logic behind this algorithm can be applied to many other artificial intelligence problems and can easily serve as a stepping stone from which many more complicated and complex AI algorithms can be developed. Although search trees and algorithms that implement heuristics have been around for nearly half a century, artificial intelligence is still in its infancy and we are only just beginning to realize its potential applications. Lines of Action, like many other games, offers a "closed domain with well-defined rules" [1] in which to test new ideas and concepts in the field of artificial intelligence.

ACKNOWLEDGMENT

The author would like to thank Mark Winands for making his impressively extensive research on the topic of Lines of Action so readily available to the general public.

REFERENCES

[1] M.H.M. Winands, "Analysis and Implementation of Lines of Action," M.S. thesis, Dept. Comput. Sci., Maastricht Univ., Maastricht, The Netherlands, 2000.

[2] M.H.M. Winands *et al*., "The Quad Heuristic in Lines of Action," *ICGA J.*, vol. 24, no.1, pp. 3-15, Mar. 2001

[3] Unknown author. (2002, Nov). *Mona and YL's Lines of Action page* [Online]. Retrieved 24 Mar 2011. Available: http://webdocs.cs.ualberta.ca/ darse/LOA/

[4] Hamed Ahmadi, *An Introduction to Game Tree Algorithms* [Online]. Retieved 1 May 2011. Available: http://www.hamedahmadi.com/gametree/#minimax

[5] Wikipedia contributors. (2010, Nov 7). *Lines of Action* [Online]. Retrieved 23 Mar 2011. Available: http://en.wikipedia.org/wiki/Lines_of_Action

[6] Wikipedia contributors. (2010, May 21). *Computer Olympiad* [Online]. Retrieved 25 Mar 2011. Available: http://en.wikipedia.org/wiki/Computer_Olympiad

[7] Wikipedia contributors. (2011, Feb 23). *Mind Sports Organisation* [Online]. Retrieved 27 Mar 2011. Available: http://en.wikipedia.org/wiki/Mind_Sports_Organisation

[8] Unknown author. (2011). *Mind Sports Olympiad* [Online]. Retrieved 27 Mar 2011. Available: http://www.boardability.com/home.php

[9] Wikipedia contributors. (2011, Mar 4). *Perfect Information* [Online]. Retrieved 23 Mar 2011. Available: http://en.wikipedia.org/wiki/Perfect_information

[10] Wikipedia contributors. (2011, Mar 21). *Zero-Sum game* [Online]. Retrieved 23 Mar 2011. Available: http://en.wikipedia.org/wiki/Zero-sum

[11] Unknown author. (2007, May 8). *Lines of Action* [Online]. Retrieved 24 Mar 2011. Available: http://www.boardspace.net/loa/english/index.html

[12] D. Graffox. (2009, Sept). *IEEE Citation Reference* [Online]. Retrieved 24 Mar 2011. Available: http://www.ieee.org/documents/ieeecitationref.pdf

[13] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

[14] G. Kaiser, C. Partridge, S. Roy, E. Siegel, S. Stolfo, L. Trevisan, Y. Yemini and E. Zadok, *Writing Technical Articles*, Retrived Jan 5, 2008 from http://www.cs.columbia.edu/ hgs/etc/writing-style.html