Exploring Optimization Strategies in Board Game Abalone for Alpha-Beta Search

Athanasios Papadopoulos, Konstantinos Toumpas, Antonios Chrysopoulos and Pericles A. Mitkas

***Abstract*—This paper discusses the design and implementation of a highly efficient MiniMax algorithm for the game Abalone. For perfect information games with relatively low branching factor for their decision tree (such as Chess, Checkers etc.) and a highly accurate evaluation function, Alpha-Beta search proved to be far more efficient than Monte Carlo Tree Search. In recent years many new techniques have been developed to improve the efficiency of the Alpha-Beta tree, applied to a variety of scientific fields. This paper explores several techniques for increasing the efficiency of Alpha-Beta Search on the board game of Abalone while introducing some new innovative techniques that proved to be very effective. The main idea behind them is the incorporation of probabilistic features to the otherwise deterministic Alpha-Beta search.**

I. Introduction

The Alpha-Beta algorithm has been applied to a variety of scientific fields [1], [2], [3], [4], partly due to its inherent generality which makes it easily adaptable to different problems. Alpha-Beta and Monte Carlo Search are two different approaches for the same type of problems. Alpha-Beta is the deterministic approach, exploiting all the available information to find the best result. On the contrary, Monte Carlo search is a probabilistic approach, which explores multiple runs to increase the chance of getting reliable results from a small subset of solutions.

Careful analysis of both of these methods often leads to the conclusion that the optimum seems to lie somewhere in between them. That is why most of the highly successful performance enhancing techniques for Monte Carlo Search, in recent years, involve exploiting knowledge based factors. At the same time, new techniques for the Alpha Beta Search are based on making estimations, which can achieve considerable performance improvement. The general idea of, almost, all of these techniques is to incorporate estimations in this genuinely deterministic algorithm.

In this paper, several successful techniques for increasing the efficiency of Alpha-Beta Search are analyzed, based on the algorithm's inherent deficiencies. Some of these algorithms were combined, yielding additional efficiency gains, while a set of new innovative techniques were implemented to further optimize the search performance.

The paper is organized as follows. Section II offers a brief overview of the game of Abalone and its rules, MiniMax and Alpha-Beta pruning algorithms, and the design of the selected evaluation function. Section III go over all the implemented techniques. More specifically, we discuss modified versions of traditional techniques, while Section IV introduces an innovative new technique and analyzes its logic. Section V presents our experimental results of the speed and efficiency tests. Finally, Section VI contains our conclusions and our suggestions for further research.

A. Papadopoulos (email: athanapg@auth.gr) and K. Toumpas (email: ktoumpas@auth.gr) are with the Aristotle University of Thessaloniki, Greece ; A. Chrysopoulos (email: achryso@issel.ee.auth.gr); P. A. Mitkas (email: mitkas@auth.gr) are with the Aristotle University of Thessaloniki and Center for Research And Technology Hellas, Greece

II. Abalone, Min-Max and the Evaluation Function

A. Abalone

Abalone is a two-player strategy board game, invented in 1987 by Laurent Levi and Michel Lalet. Examining its name's etymology, the word consists of the prefix "ab", meaning never, and the word "alone". That summarizes the main strategy of the game: "Winning against loneliness". The concept of the game is based on the popular Japanese Sumo wrestling.

Each player has fourteen marbles in his disposal distinguished by their black and white color. The goal of Abalone is to eliminate all marbles of the opponent by pushing them over the hexagonal board of the field. During each turn, a player has to move up to three consecutive marbles, either through a straight path or sideways. Pushing off the opponent's marbles can only be achieved by outnumbering the defending marbles while moving straight. As a result, the player who possesses the center of the board has a much better chance of winning the game than an opponent with his marbles spread and closer to the edge.

Another important aspect of the game's strategy is to keep a player's marbles as tightly packed as possible. Leaving a single marble away from the pack makes it vulnerable to attacks from every direction. However, if you keep all your marbles in one place, most preferably in the center of the board, you cannot be easily pushed out and it gives you the advantage of planning your attacks without the fear of being chased by the opponent.

B. Minimax and Alpha-Beta Search

MiniMax is a recursive algorithm that was originally designed for two-player zero-sum games but today it is used in a variety of different scientific fields [5]. Its name derives from the goal that every player has: to maximize its gain and at the same time minimize the opponent's.

In practice, MiniMax creates a tree with all the possible moves up to the desired depth, with the players alternating turns. The max player is always at the root. After the tree is

978-1-4673-1194-6/12/$31.00 ©2012 IEEE

complete, evaluation values are assigned to each leaf. These values should represent a reliable estimation of the game's state after each sequence of executed moves from the root to the specific leaf. The rest of the nodes of the tree are filled with the maximum or minimum values of their children as we move in a bottom-up fashion; maximum for even depths and minimum for odd ones. This process is repeated until a value is assigned to the root of the tree. That value indicates our estimation about the best possible move that can be made at the current stage of the game.

It was soon realized that MiniMax has a major performance flaw; it makes many unnecessary calculations. This drawback was overhauled by Alpha-Beta pruning, a modification of the MiniMax algorithm, designed to cut off parts of the tree that do not have to be calculated as their values will not make a difference in the evaluation [6], [7]. Alpha Beta pruning belongs to the "branch and bound" class of algorithms. It is based on the fact that during the search, a max-father node will accept only values that are higher than a certain limit: the maximum of its currently examined children (limit Alpha). Accordingly, a min-father node will have an upper limit that we call beta. Thus, for every level of the tree higher than two, an alpha-beta window is defined. This window is updated in every step and when it is closed the rest of the branches of the current sub-tree are pruned. The result is the same as using the simple MiniMax algorithm, but the performance gain can be significant, especially if the depth of the search tree is high [8], [9].

C. Evaluation Function

In Game Theory the *evaluation function* is a measure that estimates the superiority of our position at a certain state of the game. Its value depends only on the layout of the game board and the player whose turn is next. It is a static estimation, without taking into account any previous sequence of moves that has been played on the board. Consequently, it cannot unveil the player's strategy; it can only indicate the advantage we may have at a specific time-step of the game.

Despite that, the design of the evaluation function is very important to the efficiency of the Alpha-Beta algorithm. A very sophisticated design that would guarantee a fairly good estimation of the state of the game would increase the accuracy of the value. However, due to the increased computational complexity, the speed of the evaluation would be very low, compromising the depth of our search tree for a certain time interval. Hence, a balance between accuracy and complexity has to be achieved.

The 'perfect' balance, however, depends on many factors. For tree strategies with high expansion rates and moderately pruning Alpha-Beta algorithms, a sophisticated evaluation function would be preferable, minimizing the sacrificed depth levels. On the other hand, for strategies with moderate or low expansion rates and highly efficient pruning algorithms a rough but speedy estimation would be the optimal choice. Abalone has an expansion rate of approximately sixty nodes per level, which can be considered moderate [10], [11], [12]. So, our implementation of the evaluation function is designed to be fast but still trustworthy enough. The logic of the estimation of the evaluation function implemented takes into account several parameters:

Number of marbles

The number of marbles each player has is a strong indicator about the current state of the game. The effect of the number of a player's marbles on the evaluation function varies with the number of marbles its opponent has and is used to control the player's aggressiveness.

Distance from the center

Abalone is mainly a defensive game. As many abalone World Champions say, "capturing the center is one of the most important factors for winning the game". This parameter does not represent the Euclidean distance of each marble to the center of the board, but the Manhattan one, which is the minimum amount of steps you have to make in order to reach the center.

Coherence

As the game's name itself states, keeping your marbles coherent not only keeps them safe, but also gives them the ability to attack effectively. This is calculated as the number of neighbors each marble has, giving also an extra bonus in three marbles in-line formations.

Formation break

Placing your marble among the opponent's marbles has many advantages. Not only your marble cannot be attacked, but also the formation of your opponent's marbles gets split in half. Thus, they are less capable of attacking and become much more vulnerable to being captured. This parameter is calculated as the number of marbles that have opponent marble in opposite sides.

Single marble capturing danger

When one of the player's marbles touches the edge of the board, while having neighboring opponent marbles, there is a possibility of being captured. If there is only one, the reduction is minor. If there are more than one opponent marbles the danger is imminent and the reduction in the evaluation is high.

Double marble capturing danger

The same goes when two of the player's marbles touch the edge of the board and have more than one opponent marbles on the opposite side. This time the evaluation reduction is even higher than before, because there are more than one marbles in danger.

It must be noted that only the first one is the only *material* feature parameter of this game. The rest belong to the *spatial* feature parameters family.

III. Analysis and Improvement Of Known Techniques

Quiescence Search

This method is used to alleviate the consequences of the Horizon Effect by extending the moves' search in special occasions [7]. Searching all possible moves at the same depth is not always reliable. For example, if a leaf of the decision tree gives the opportunity to capture an opponent's marble, the evaluate function will give a high result. However this value is not reliable, because by making that exact move we may provide our opponent the opportunity to capture one of our own marbles. Especially in games like Abalone, this happens very often, considering that in order to capture an opponent's marble your marbles have to be near the edge of the board, making them prone to an attack.

The method of Quiescence Search provides a solution to this problem by extending the maximum search depth *Dmax* by a constant q, when the move that leads to a leaf of the tree is a capturing one. Thus, there can be branches reaching q levels deeper than the nominal depth of the tree. Increasing d (the depth at a given time) over a certain threshold is not as efficient, because the improvement does not count up for the overhead.

Usual Minimax/Alpha-Beta Pruning

```
if d >= Dmax then
   Stop/Return
else
   Go Further Down The Tree
```

Quiescence Search

```
if ((d >= Dmax)
AND White Pieces Number Stays The Same
AND Black Pieces Number Stays The Same)
OR (d >= Dmax+q) then
      Stop/Return
else
      Go Further Down The Tree
```

A human player applies this technique intuitively. He spends more time analyzing the moves that involve action and less in the quiet ones. Using this method in Abalone with q=2, the performance of the algorithm decreases on average by about 20%, but its results are much more reliable. By trading off speed for quality, the algorithm makes better decisions with a slight decrease in speed, resulting in significant overall gain.

Non-Linear Aspiration Windows

Aspiration Windows are generally used in order to decrease the Search Space in an Alpha-Beta Pruning algorithm [13]. In many occasions, a sequence of moves leads either to a very good result or a very bad one. If, for example, during the move search, a child-node of a max-father that has an extremely high value is discovered, it could be selected straight away, ignoring the rest of its siblings, guessing that this value is large enough for surpassing the rest of the child-nodes of the max-father. In most cases, this guess is correct and the time saved is significant. Even if this guess turns out to be wrong, it ends up with a value fairly close to the optimal one.

This technique can be generalized by setting a search "window" for each level of the tree. The upper and lower limit of that window are defined as the highest and the lowest reasonable value that a node at this level can have. If a node's value falls outside that window, depending on the father type (max or min), we can interrupt our search and return that same value for this node. The overhead of this method is the call of the evaluation function for every node of the tree and not just for the leaves as in the standard Alpha-Beta pruning. This value is essential as far as the estimation of the current state of the game is concerned.

A human player uses this kind of logic all the time. Some moves can be easily considered good or bad with very little thought. Therefore, he can spend his time thinking about the more balanced moves that may turn out to be, in the long run, more profitable. Even though this may be a low probability, it turns out to be very effective.

The implementation of this technique exploits the use of the Alpha-Beta pruning, amplifying the window "cuts" with the alpha-beta "cuts". Thus, a and b can be derived as follows:

$$estimation = evaluate(game) \tag{1}$$

$$a = max(a, estimation - windowsize) \tag{2}$$

$$b = min(b, estimation + windowsize) \tag{3}$$

The window size should be different for every level of the tree. At lower levels, the window has to be narrower because the sub-trees of the children-nodes are less deep. However, the window should be much wider at higher levels of the tree, considering that a sequence of fairly good moves can return very high value, meaning that a value that may be considered "extreme" for a certain level of the tree, can be quite reasonable for another level.

While designing the Aspiration Windows, the best trade-off between speed and quality of search has to be found. The narrower these windows are, the faster the search is completed for a certain depth of the tree, sacrificing part of the reliability of the result. On the other hand, the increased speed of the search, allows searching into greater depth, thus increasing the quality of the results in the same time interval. Therefore, the desired window width is the one that maximizes the quality of the search. The most commonly used procedure for the determination of the windows' width is their gradual narrowing up to the point that the Aspiration algorithm can marginally beat an one-level deeper Min-Max Search. This way, the low degradation of the quality of the player is guaranteed.

The usual approach to Aspiration Windows is to calculate the width of the search window using a linear equation with respect to the level of the tree. The process can be graphically represented as an inverted jagged cone, shown in Fig. 1, whose characteristics are defined by each level's upper and lower limits. The drawback of this implementation is that

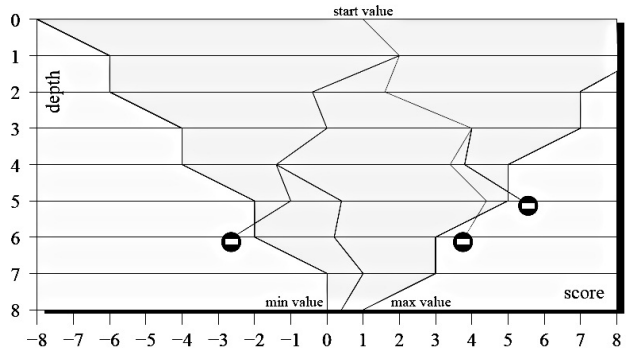

Fig. 1. A linear Aspiration Window. As we move further down the tree the window becomes smaller. However, this reduction is done linearly, resulting in a "search cone" that is wider than optimal at the upper levels and narrower than optimal at the lower ones [14].

the search windows become unnecessarily wide in the upper levels, consuming time in futile searches, and too narrow at the lower levels, resulting in serious errors. This behavior is contrary to the main idea of Aspiration Windows, which is to discern what is worth searching and what is not. The calculation of the window for every level is defined by the following equation:

$$windowsize = (treedepth - currentdepth) * constant \quad (4)$$

In our implementation, each level window is treated independently. Finding the ideal combination of these widths leads to a schema similar to a narrowing well. The performance increases up to tenfold compared to a simple search in the same depth, sacrificing only a reasonable amount of the quality of the result. This is accomplished by identifying the nodes that contain useful information and avoiding the ones which do not.

TRANSPOSITION TABLE

A common method that is applied when a problem consists of overlapping sub-problems is *Memoization* (making a memo). As the name indicates, this method involves storing the result of solved sub-problems in order to use it, in the future, without the need of making the same calculations all over again ([15], [16]). Especially in games like Abalone, which has a board composed of similar pieces, overlapping nodes or even entire sub-trees can often occur. Thus, a hash table is created to store the results of the evaluation function for each distinct layout of the game board.

Firstly, a unique number needs to be assigned to each different state of the game board. For board games, the most efficient method to accomplishing this is Zobrist Hashing, which involves creating a random 64-bit number for every color and place a marble can have on the board. Then, to calculate the unique number of the present state, we just XOR the numbers of the marbles that are at these places, resulting in another 64-bit that characterizes uniquely the current layout of the game board. The probability for this number to be the same for two or more different states of the board (collision) is calculated as follows:

$$P = 1 - e^{-10,000,000/2*2^{64}} = 2.710501754 * 10^{-6}$$

Therefore, we consider this key to be reliable and ignore the possibility of collision.

Zobrist Hashing is incredibly fast because XOR calculation takes almost no time to complete. Furthermore, the XOR operator has the attribute of being self-inversive. Thus, each state of the game board does not have to be calculated all over again. We can, for every marble that is moved, just use the XOR operator on the 64-bit number of the source (extracting that information) and the destination (inserting that information) to the key of the father-node.

The problem that arises is that the hash table must have an entry for every different 64-bit key, and thus the size of $2^{64} = 18.446.744$ TeraEntries. Consequently, another level of hashing is essential in order to create the pointer-key of the table. A layered view of this logic is shown in Fig. 2. Selecting a table with 4MegaEntries, we use the mod(2^{22}) function to produce the hash key indices of the table. These keys are comprised of the last 22 bits of the 64-bit number and have a probability of collision:

$$P = 1 - e^{-10,000,000/2*2^{22}} = 0.696415$$

Since this 22-bit key is not so reliable at all and to avoid any unwanted results, we store the entire 64-bit number together with the result of the evaluation function, so as to match it every time we want to use this value (Collision Resolution).

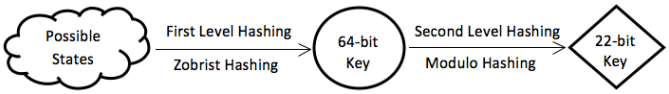

Fig. 2. The hashing stages. For each possible state of the game a reliable 64-bit key is assigned. Then for all these keys a less reliable 22-bit key is assigned. The measure of reliability is the probability of collision.

Hence, every time we have to call the evaluation function, we calculate the 64-bit and 22-bit keys and then check if the corresponding entry that the 22-bit points to contains the same 64-bit key. If it does, we use the stored value. Otherwise (second-level hashing collision), we call the evaluation function and store the result together with the 64-bit key. Every entry of the table can have a third variable, which will act as a flag, indicating that the value has been used at least once before. This way, when a second-level hashing collision occurs, the stored value is replaced with a new one, accordingly. The purpose of this technique is to keep in store the more frequent moves and replace the ones that have never been used.

Furthermore, Transposition Tables can store not only the results of the evaluation function, but also the values of entire sub-trees. Even though, in an Alpha-Beta search, the values assigned to the non-leaf nodes are not exact, but upper or lower bounds, the information that these numbers carry is still worth storing. Thus, we add a fourth variable to our entries,

to indicate whether the stored value is exact, lower or upper bound. In the last two cases, we cannot avoid searching the sub-tree, but we can use this value in order to make our a-b window narrower and consequently speed up our search.

All in all, Transposition Tables have great impact on the performance of the algorithm. With the use of a table with 4MegaEntries the search becomes approximately three times faster. This is due to the fact that the overhead of the calculation of the keys is negligible, whereas the gain from avoiding calling the evaluation function is significant.

ITERATIVE DEEPENING

When a move has to be decided within a specific time frame, predetermining the tree's depth is not a wise action. It is better to complete the search at a shallower, thus safer depth and then steadily increase the depth, while checking the remaining time. Just before the time limit is reached, the search for the best possible move at the increased depth is aborted and the move from the previously completed depth search is retrieved [17], [18]. By using this technique, we fully exploit our given time frame whilst safely acquiring a reliable move.

As far as the calculation cost of this technique is concerned, the loss is really minimal for, mainly, two reasons: a) due to the relatively high branching factor of the tree, the cost difference between directly searching at a specific depth and calculating the previous level in addition to the specified is minor, b) by using a Transposition Table, many of the evaluation function's values of the shallower search are stored and can be retrieved while searching deeper. In our experiment with Abalone, the use of Iterative Deepening combined with Transposition Tables has consistently reduced our code's execution speed by about 25%.

It must be noted that an unexpected effect occurs by combining Transposition Table with Iterative Deepening; the algorithm does not always play the same way. This is caused because the numbers generated for the Transposition Table are random. Therefore, with each run of the algorithm different collisions occur (at the second level hashing) resulting in a Table with completely different entries. These differences can sometimes prove to be useful (e.g. used more often) and others not so. For that reason, the search speed at a specific depth is not fixed; it varies depending on the Table's entries. In case there is a time limit, it is possible that the algorithm will not make the same moves each time it is executed.

IV. COMBINED MOVE ORDERING

An innovative technique, which turned out to be the most effective, is named Combined Move Ordering (CMO) and is presented for the first time in this paper. It is true that the more well sorted the children of a node are, the more efficient the Alpha-Beta pruning becomes. For example, if we knew in advance which is the best move for every level, pruning would become optimal and only two nodes for each sub-tree of the root would be examined, one that would give the desired value, and one that would close the $[a,b]$ window.

Even if the children of every node are not perfectly sorted, the performance gain could be quite significant. If they are sorted by estimating which move is expected to be better, then the probability of earlier pruning is increased. This means that at every level of the tree, before delving deeper, the child-nodes can be sorted by some kind of estimation metric. Especially in games that produce trees with relatively high branching factor (such as Abalone), this method is extremely effective.

The gain from sorting the child-nodes is more significant at higher levels of the tree, because pruning of deep sub-trees can happen earlier. At lower levels, this gain is smaller, due to the shallowness of the pruned sub-trees. Thus it is reasonable to divert more effort on higher levels of the tree, to get a more proficient estimation. The tree was divided to three distinct regions and for each one a different method was implemented for sorting the child-nodes. A layered view of this logic is shown in Figure 3. The three techniques (a) Evaluate Sorting for higher levels, b) History sorting for middle levels and c) Marble Ordering in lower levers) are explained below.

A. Evaluate Sorting

At the first three levels, the evaluation function is calculated for every child of the father-node and then sorted by their value (ascending sorting for the min-fathers, descending for the max-fathers). This estimation is fairly accurate and pruning now occurs at 90% of the cases in the first 6 kids. The overhead of this method is that, before doing the actual search, the evaluate function has to be called for every child-node of a father, even for the ones that are going to be pruned. This may seem to put a heavy burden on our algorithm, but, in reality this is not the case. In essence, pruning at the higher levels of the tree is traded for pruning at the lower ones. Thus, at the lower levels, where our tree becomes extremely wide, more intensive pruning is being performed.

B. History Heuristic Sorting

After the first three levels, Evaluate Sorting is not as effective, because the computational cost is higher, due to the fact that the tree becomes wider, while at the same time the performance gain is smaller, since the pruned trees are shallower. Consequently, for levels four and five, we define a different region, in which sorting is based on how many times a move has caused pruning (at the same level of the tree). This method is known as History Heuristic and it is an evolution of the Killer Moves Heuristic. In the later, only one or two moves that caused pruning (killer moves) are stored and these are examined first.

The History Heuristic involves holding a table with an entry for every move and level that stores the number of prunings (kills) this move has caused. Every time a kill is made by that move, we increase this counter by one. Thus, the sorting of the child nodes is based on the number of kills the move that leads to them has caused. A move that has caused a kill for a specific layout of the game board is most likely to cause another kill for a similar layout (at the not so deep levels of the

tree, cousin-nodes have similar board layouts). For example, a capture move is an exceptional move, even if the opponent moves a distant marble somewhere on the board. Although this is not always true, a move that has caused many kills has a high probability to be a good move for many different formations of the game board.

The performance gain of CMO is significant, because the sorting is quite accurate and the computational cost relatively low. The overhead is mainly the cost of the sorting, which must be performed many times, due to the tree's width at these levels. The sorting algorithm utilized in our approach was Quicksort, which is ideal for inputs of that size.

C. Marble Ordering

Below the fifth level we do not utilize any of the above methods, because the overhead increases considerably and the board formations are markedly different. For the lower levels a third area is defined, in which sorting is based on how many marbles are moved and what kind of move they make. Sorting take place according to the following descending order:

1) Three marble capturing move
2) Three marble attacking move
3) Three marble in-line move
4) Three marble broadside move
5) Two marble capturing move
6) Two marble attacking move
7) Two marble in-line move
8) Two marble broadside move
9) One marble move

Although this is a rough estimation, it turns out to be very effective. Its overhead is almost zero, due to the fact that this ordering can be integrated in the function that finds all the possible moves for a player. Such function handles all these cases independently, storing them in different tables that can be merged in the correct order in the end.

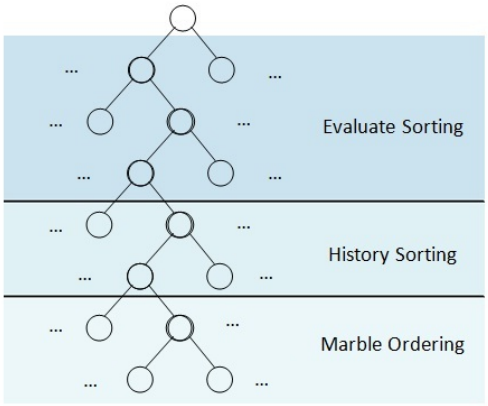

Fig. 3. Combined Move Ordering

D. Iteration Sorting

Taking advantage of Iterative Deepening,CMO can become even more effective by exploiting the information from the search at a shallow game tree for faster search in the deeper ones. For example, at the end of the four-level search, we have calculated some of the evaluate functions of the leaves. These values can be used during the five-level search, to improve the sorting of our the fourth-level nodes. This technique is called Iteration Sorting. Due to extensive pruning, the number of evaluated leaves is really small. Nevertheless, the information that can be harvested from them is valuable and can be used effectively in the lower two regions defined in our tree (Figure 4).

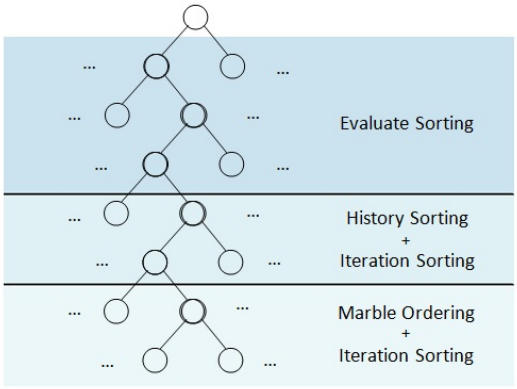

Fig. 4. Advanced Combined Move Ordering

For example, after the four-level search is completed, and during the five-level search, for every node at level three, we first sort all of its child-nodes based on the number of the kills they caused (History Heuristic) and then we examine for which of them the evaluation function has been calculated in a previous search (their values are in our Transposition Table). Afterwards, we move one third of the best of them to the front, sorted based on their value. Also, we move one sixth of worst of them to the back. For the rest of them, we only change their relative position based on their value. An example of this two-level sorting is shown in Figure 5.

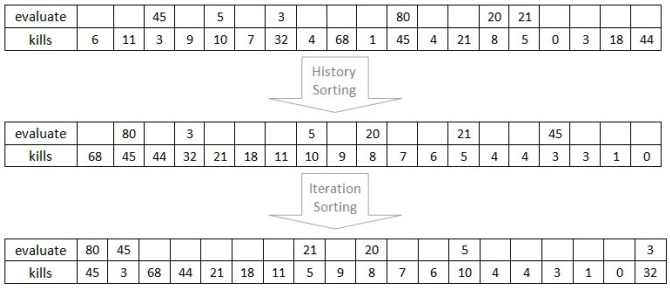

Fig. 5. Combining Different Sorting Methods to Improve Estimation

The same method can be applied in the third region of our tree. So, we managed to combine two sorting criteria for better estimation. We chose to move only one sixth of the worst moves to the back because in case one of them turns out to be the only good one, being at the absolute back will slow down our algorithm significantly.

All in all, sorting takes place in three different ways depending on the level of the node under examination. It is really

TABLE I
KNOWN ALGORITHMS COMPARISON

Algorithm:	Alpha-Beta	Quiescence	Killer	Aspiration
Depth 5:	5.574.269	8.239.915	6.373.827	1.007.063
Depth 6:	24.000.073	107.733.517	25.000.317	4.000.015
Depth 7:	399.181.277	526.422.388	391.203.153	19.254.405

TABLE II
ADVANCED ALGORITHMS COMPARISON

Algorithm:	Aspiration	+Comb. Move Ord.	+Trans.Tab.& Iter.Deep.
Depth 5:	1.007.063	577.097	400.753
Depth 6:	4.000.015	2.523.015	1.004.517
Depth 7:	19.254.405	8.452.204	2.921.523

important that a bigger investment must be made at higher levels; because greater profit is expected, while much smaller profit is expected from the lower ones. A human player would do exactly the same; first think about the moves that seem profitable on short term, because they are the ones that are most likely to be good on the long term (Greedy Logic). Most of the 'recipes' presented were known before and considered ineffective [19]. But they prove to be extremely effective when used in each of the tree regions they are more suitable for. Thus, splitting the search tree into different regions and using different ordering technique for each region is the key of this method.

V. EXPERIMENTAL RESULTS

Experimental testing involved many runs against various opponents in order to check the consistency of the newly designed algorithms. The opponents that our algorithms faced had different alpha-beta and evaluation function implementations. We also made runs against human players in order to test the predictability of the algorithms' logic. Testing our implementation against a diverse spectrum of opponents, we determined both the quality and the performance consistency of our algorithms.

The result of our experiments expressed as the total number of nodes examined for each of the algorithms presented earlier are listed in Tables I and II. The data, also illustrated in Figures 6 and 7, compare the speed of each algorithm, without considering other parameters (choices made by the algorithm, etc). Tree searches of depth five, six and seven were tested, in order to show the implementations capabilities in dynamic environments. For deeper tree searches, the time for a mainstream computer to finish a game is prohibiting, even though at higher depths the techniques analyzed would have an even greater impact on the percentage of the performance boost. The opponent chosen for these experiments is a heuristic opponent with the same evaluation function as the one utilized in our implementation. The huge differences in examined nodes dictated the use of a logarithmic scale to plot the results.

At depth seven the algorithm becomes approximately 25% slower when it encapsulates the quiescence technique. However, the quality increase is significant enough to account for

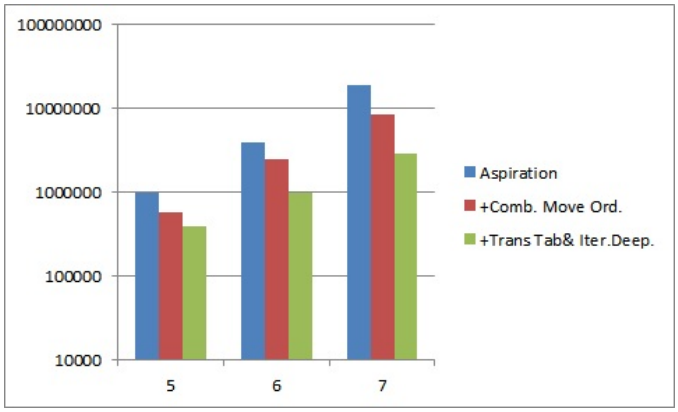

Fig. 6. Nodes examined by each known algorithms at three different depths.

Fig. 7. Nodes examined at different depths after the integration of all the different techniques in the search engine of the Non-Linear Aspiration Window algorithm.

that performance drop. The difference is substantial enough compared to the speed loss, as the player becomes more predictive at the large-scale conflicts outcome. On the contrary, the Killer Move Heuristic, after the fifth level of the tree, is not efficient at all.

Moving on to the next algorithm, the implementation of Non-Linear Aspiration Windows massively increases the algorithm's speed. Our player becomes almost twenty times faster, while losing some of its quality due to the "guessing" procedure mentioned in III. However, the decrease in the intelligence of the algorithm is minor compared to the performance gain, since our criterion for the sizing of the windows was the ability to beat an one level shallower Min-max algorithm with the same evaluation function. With this kind of "node economy", you can search at least two levels deeper in the same time, and thus increase the quality again to a greater proportion. In the end, the increase in the efficiency of the algorithm is more than significant.

After encapsulating the Combined Move Ordering in the player's implementation, the result is approximately 250% faster at a seven level search. The region-splitting of the tree and the use of a different child-ordering technique in each of them manages to make great "node economy" for deep searches. With the addition of the Transposition Table

and Iterative Deepening techniques, the algorithm runs almost 300% faster, due to the reuse of evaluation values acquired both earlier at this search and in previous shallower searches.

Further testing that proves the quality of our player have been carried out, but isn't presented here due to lack of space. These tests show that this significant performance enhancement is being done with a minimal quality degradation.

VI. Conclusion and Further Research

In this paper, we investigated the design and implementation of a highly efficient MiniMax algorithm for the board game of Abalone. We explored several techniques for increasing the efficiency of Alpha-Beta Search and introduced some new innovative ones that proved to be very effective. The goal of almost all of these techniques is to reduce the redundant searches of nodes. Selecting the nodes that "deserve to be searched" from the ones that do not carry any information is the heart of the logic of these techniques; being able to estimate the values of the child nodes before you calculate the evaluation function for them.

Implementing all these techniques into the traditional Alpha-Beta Search increased the algorithm's performance up to one hundred thirty times. The speed-up lies mainly in the efficiency of the search engine. For example, searching a tree with depth seven results to an average of 3 million examined nodes. The full tree comprises a total of $60^7 = 3 \cdot 10^{12}$ leaves, meaning that only 1 in about a million of the total nodes is actually examined, rendering the algorithm approximately one million times faster than the traditional Minimax.

Any kind of "effective guessing" that can be integrated to the Alpha-Beta search increases its efficiency dramatically. However, the performance overhead of this "guessing" has to be lower than the performance gain acquired by the extra pruning that it will produce.The appropriate combination of the different techniques plays a significant role to the overall efficiency of the algorithm.

For further research, a good idea would be to implement sibling prediction [20] in order to narrow the search windows more effectively. The combination of aspiration windows and sibling prediction can further improve the overall performance of the algorithm, by providing a better insight of the expected evaluation values that the other children-nodes will have. As a result, a more effective "adaptive narrowing" of the aspiration windows may be achieved. A careful implementation of the Null-Move technique [21], [22] could also have a significant impact on the performance of the algorithm. Applying it only to certain levels of the tree can protect the search from major quality losses.

Finally, a more sophisticated design of the evaluation function could boost the efficiency of these techniques. An evaluation function specially implemented for the logic of the specific game (or whatever application of the Alpha-Beta search) can have a considerable impact. Whereas the design of the Min-Max performance enhancement techniques has to be general, the design of the evaluation function should be as specific to the game as possible, since that is the only point that the logic of the specific application can, and must, be imported into the algorithm[14]. Further testing between different evaluation functions should be carried out, in order to find out the 'needs' of our algorithm.

References

[1] T.-P. Hong, K.-Y. Huang, and W.-Y. Lin, "A genetic minimax game-playing strategy," in *Evolutionary Computation proceedings, 1998. IEEE World Congress on Computational Intelligence, The 1998 IEEE International Conference on*, May 1998, pp. 690 –694.

[2] C. Ong, H. Quek, K. Tan, and A. Tay, "Discovering chinese chess strategies through coevolutionary approaches," in *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, April 2007, pp. 360 –367.

[3] D. Simon, "A game theory approach to constrained minimax state estimation," *IEEE Transactions on Signal Processing*, vol. 54, no. 2, pp. 405 – 412, February 2006.

[4] C. Charalambous, "Stochastic nonlinear minimax dynamic games with noisy measurements," *IEEE Transactions on Automatic Control*, vol. 48, no. 2, pp. 261 – 266, February 2003.

[5] T. A. Marsland and Y. Bjrnsson, "From minimax to manhattan." in *Deep Blue Versus Kasparov: The Significance for Artificial Intelligence'97*, 1997, pp. 31–36.

[6] J. Schaeffer, "The history heuristic and alpha-beta search enhancements in practice," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, pp. 1203–1212, 1989.

[7] A. K. Griffith, "Empirical exploration of the performance of the alpha beta tree-searching heuristic," *IEEE Transactions on Computers*, vol. 25, no. 1, pp. 6–11, 1976.

[8] A. L. Samuel, "Some studies in machine learning using the game of checkers. ii recent progress," *IBM Journal of Research and Development*, vol. 11, no. 6, pp. 601–617, 1967.

[9] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial Intelligence*, vol. 6, no. 4, pp. 293–326, 1975.

[10] P. Chorus, "Implementing a computer player for abalone using alpha-beta and monte-carlo search," Master's thesis, Dept. Knowledge Eng., Maastricht University, Aahen Germany, 2009.

[11] E. Ozcan and B. Hulagu, "A simple intelligent agent for playing abalone game: Abla," in *Proc. of the 13th Turkish Symposium on Artificial Intelligence and Neural Networks*, 2004, pp. 281–290.

[12] N. Lemmens, "Constructing an abalone game-playing agent," June 2005.

[13] H. Kaindl, R. Shams, and H. Horacek, "Minimax search algorithms with and without aspiration windows," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, no. 12, pp. 1225–1235, 1991.

[14] T. W. Oswin Aichholzer, Franz Aurenhammer, "Algorithmic fun - abalone," 2002, institute of Theoretical Computer Science, Graz Univ. of Technology, Graz, Austria.

[15] J. Veness and A. Blair, "Effective use of transposition tables in stochastic game tree search," in *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, april 2007, pp. 112 –116.

[16] 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*, vol. 13, no. 5, pp. 447–459, 2002.

[17] V. S. Gordon and A. Reda, "Trappy minimax - using iterative deepening to identify and set traps in two-player games," in *2006 IEEE Symposium on Computational Intelligence and Games*, May 2006, pp. 205 –210.

[18] X.-Z. Wang, Y.-L. He, P. Su, and W.-L. Li, "Two-ply iterative deepening in chinese-chess computer game," in *Machine Learning and Cybernetics, 2009 International Conference on*, vol. 4, July 2009, pp. 2020 –2026.

[19] N. R. Sturtevant, "Multi-player games: Algorithms and approaches," 2003.

[20] J. W. Carolus, "Alpha-beta with sibling prediction pruning in chess," Master's thesis, Faculteit der Natuurwetenschappen, Wiskunde en Informatica, Univ. of Amsterdam, Netherlands, 2006.

[21] O. David-Tabibi and N. S. Netanyahu, "Verified null-move pruning," *ICGA Journal*, vol. 25, no. 3, pp. 153–161, 2002.

[22] O. David-tabibi and N. S. Netanyahu, "Extended null-move reductions," in *In Proceedings of the 2008 International Conference on Computers and Games*. Springer, 2008, pp. 205–216.