Parallel Intelligent Game-Playing Algorithms for Rhythmomachy

Zach Anderson, Colin Dawson, Zach Firth,
Matt Jennings, Adam Thimons, Pius Uzamere II

Abstract

Our team developed an intelligent game-playing program for Rhythmomachy to run on one of the world's fastest parallel supercomputers, the CrayT3E. A game engine and user interface were developed. An artificially intelligent game player was created which used an exhaustive MINIMAX search tree and a heuristic board evaluation function to imagine possible moves and choose the most intelligent. The team successfully produced a game player with a near perfect success rate against experienced human opponents.

I. Introduction

The year is 1100. Deep within the dark recesses of a French monastery, a fierce battle is brewing, although not one of fists and swords, but one of numbers and proportions. A black robed monk hovers over a board, brow furrowed. His opponent leans back in his chair and sips from a goblet of wine while nervously strumming his fingers upon his leg, trying not to reveal his unease. What are these two playing? They are mulling over a game which was, at one time, more popular than chess: Rhythmomachy.

In recent years, computer scientists have had much success in creating artificially intelligent players for a variety of games. Just recently, IBM's Deep Blue computer defeated the reigning Grandmaster of chess, Garry Kasparov, at his own game. It is curious that Rhythmomachy, whose popularity once surpassed that of chess in intellectual circles, was left by the wayside in this flood of computational game research. It was our team's goal to give Rhythmomachy its rightful place in the field of Artificial Intelligence research.

II. Research

A. Origins of the Game

Although Rhythmomachy has existed for nearly 1000 years, its origins are unclear. It was once believed that the great mathematicians Boethius and Pythagoras invented the game. However, most scholars now believe that a monk named Asilo invented Rhythmomachy. The oldest written evidence of the game was found in a paper from 1030 A.D. describing the game. It states that Rhythmomachy was invented as an educational tool for teachers who were trying to instruct students in the number theory of Boethius and Pythagoras. Until about 1100, the game proliferated for the most part throughout France and southern Germany where the game's rules and structure were solidified. During the twelfth century and onward, the game spread to other regions of the world.

The rules of Rhythmomachy were originally much more complex than those seen in the PGSS computer based version. The old rules, translated by William Fulke in 1563, include six separate ways

to win, and over 80 pages of text. These rules describe a game that was not only very complicated but one that could literally take days or months to finish.

The new version of Rhythmomachy was simplified by reducing the methods of winning from the six or seven classes of victories in the original to one: a player must capture a certain number and point total of pieces. In the original version of Rhythmomachy, the main goal of the game was not to capture your opponent's pieces. The real goal of the game was to position pieces in arithmetic, geometric, or harmonic proportion on the opponent's side of the board.

Our team modified Rhythmomachy to be capture-oriented for several reasons. A game that focuses on captures is generally more captivating and exciting to human players. Entertainment value was a strong factor affecting the team's decisions. For programming purposes, having such a complex goal can be a problem. For a computer, it is significantly easier to have several sub-goals (e.g. capturing a certain piece) that eventually lead to a win, than to have one long-range goal to which a player is aspiring in order to win. The program has practical limitations on how many moves it can calculate in advance -- the team's new version of Rhythmomachy utilizes a four-level game tree. This tree would be highly inadequate if the purpose of the game were to reach a goal that ultimately entails making nine moves, for instance.

One reason that the original Rhythmomachy took so long to complete was the complexity of the playing pieces. Each playing piece was one of three shapes, and ranged in value from 1 to 361. This would not be practical in the new version of the game. Due to the arithmetic nature of the game, several pieces would be virtually untouchable to the opponent, and at the same time rather useless to the player with the piece. While these several pieces were important in the original game with its multiple winning methods, they are less useful in the PGSS '99 version of the game.

Some other factors that prolonged the game were the large 8 by 16 size of the board, and the number of pieces allotted to each player. Each player controlled 24 pieces. Having to keep track of what could become 30 or more pieces at one time on such a large board slowed the game down to a crawl, and forced players of the classical version to take very few risks.

B. New Rules

1. Board Configuration

The first difference between the original Rhythmomachy and the new PGSS version is the board configuration. Instead of the original line formation, the pieces are now arranged in a plus sign across the board. The new board is shown in Figure 1.

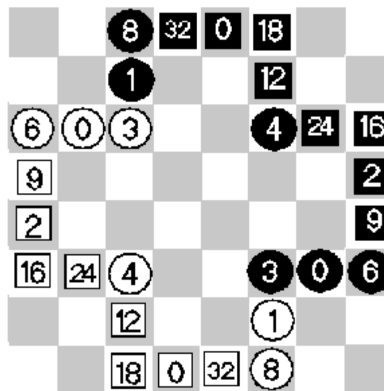


Figure 1: The New Board Configuration

2. Movement of Pieces

In Rhythmomachy Version PGSS99, there are two types of pieces: circles and squares. Each piece is double-sided. Both sides are the same with the exception that one side is black and the other white. Round pieces may move one space either orthogonally or diagonally. Square pieces move similarly to circles, but move two spaces each turn. They can move either two spaces in a straight line orthogonally or diagonally, or move one space orthogonally and one space diagonally in the manner of a chess knight. The resulting range of motion for a square is the edge of the five by five square centered at the piece.

3. Capture

Perhaps the most important and confusing aspect of capture in Rhythmomachy is the fact that a piece doesn't actually move onto an enemy piece to capture it. In chess a capture is made by moving a piece onto the square of the opponent. In Rhythmomachy, the position must only be set up. That is, as soon as a player is in position to jump on a piece in the *next* move, he captures it. The result is that a player must pay much more attention in Rhythmomachy than in chess. By the time a piece would be "in check," the capture has already taken place. Due to this, capturing in Rhythmomachy is significantly more dramatic than capturing in chess. In Rhythmomachy, it is possible to make multiple captures with a single move.

The other aspect of capture unique to Rhythmomachy is the number of pieces used. While chess and checkers require only one piece to perform a capture, Rhythmomachy requires a player to have a combination of two pieces, both in position, in order to take an opponent's piece. When two pieces are placed in a position such that both of them endanger one of the opponent's pieces, the opponent's piece is captured provided that the value of the piece to be captured is arithmetically related to the two pieces performing the capture. That is, it must be the sum, difference, product, or quotient of the two capturing pieces.

If a player makes a move that causes his piece to be endangered by two of his opponent's pieces, a capture does not immediately take place. If the opposing player wishes to actually capture the piece, she must find another square to which she can move one of her pieces such that she is still able to

capture the endangered piece. The result is that a player can never lose a piece as a direct consequence of his own move.

4. Turning the Enemy

Capturing an enemy piece does more than take it off the board – it actually begins to work for the capturing player. When a player captures an enemy piece, he flips the piece over so that his own color is facing up. Rhythmomachy pieces are, for this reason, double sided. On the computer, the color is similarly reversed. The captured piece is placed as close to the player's starting corner as possible in such a position that the piece does not immediately give him a capture. The piece now belongs to the capturing player until it is recaptured. This feature is quite important because it tends to shorten the game. Since captured pieces are not only lost by one player but also gained by the other, the game slowly unbalances once someone gets the upper hand. This places added importance on the first few moves.

5. Victory

In order to win the PGSS version of Rhythmomachy, a player must perform a set number of captures. The values of the captured pieces must also add up to a predetermined number of points. This combined requirement helps to compensate for the greater utility of lower numbered pieces by placing a greater point value on higher numbered pieces. In the computer program, these numbers are currently set at 4 captures and 25 points, but are easily adjustable as desired for a longer or shorter game.

C. Search Trees

The Rhythmomachy game player uses an exhaustive search tree. With this method, the program envisions all possible moves for itself, as well as all possible responses by the opponent. Depending on the depth of the tree, the program will search a given number of moves in the future. In order to do this it must run a loop for every piece on its side of the board. Within this loop it looks at each legal move for the piece in question. For each legal move the program stores a new board configuration in its memory. The recursive function continues to search through future possibilities for subsequent turns, until it reaches the last level of the tree, storing every potential board from the current turn to several moves ahead.

D. Branching Factor

In theory it is possible to have a "perfect" search tree. Given enough levels, the computer could search forever until it finds a direct path to a win. However, in a game such as Rhythmomachy, the branching factor is so massive that the time required for even the world's most powerful computers to perform such a search would be on the order of years. Consider the following: There are fourteen pieces available to move each turn. (This is an average – there will sometimes be more or less pieces depending on net captures.) Each piece can have up to sixteen possible moves. (Circles have at most 8.)

Each new level on the search tree increases the search time exponentially by a conservative estimate of at least a factor of fifty given the realistic limitation of move possibilities for each piece. For example, to search one move into the future would require fifty calculations. However, to perform a two-level search would require the computer to perform the same fifty calculations on each of the fifty new board configurations that it discovered. This would take a total of 50^2, or 2500 calculations. Similarly three levels would take 125,000 calculations, four levels 6,250,000 calculations, and so forth. Searching continuously until a win is found could take altogether, depending on the piece and point goals set at the beginning of the game, up to or surpassing twenty or thirty moves. The computer would then have to perform on the order of 50^{25}, or about $3x10^{42}$ calculations. Some of the fastest computers can perform a high-end estimate of roughly 900,000,000 calculations per second. This may seem fast. However, if we were to do the arithmetic, a "perfect" search tree would take a fast supercomputer $3x10^{33}$ seconds, which equates to $1x10^{25}$ years, to calculate each move. For comparison, our universe has been in existence for approximately $1.2x10^{10}$ years. Needless to say, this would most likely cease to provide any enjoyment for the player. Hence rises the necessity to have a short search tree and a heuristic evaluation function to choose the path most likely to lead to a win.

E. Heuristic Evaluation

The word heuristic is most commonly defined as "the study of the methods and rules of discovery and invention." The word is derived from the Greek verb *eurisco*, meaning, "I discover". In the context of Computerized Game Playing, a heuristic evaluation function is a method used to calculate the relative strengths and weaknesses of each possible move sequence on a search tree. At the final level of the search tree, the evaluation function is called at each node, or potential board configuration. An overall numerical value for a given board is calculated by combining the individual numerical values for various game parameters. These parameters are given relative weights according to their importance toward victory. For example, the highest value is given to a win. The next most desirable condition in the game would be a capture. The hierarchy continues with less and less important parameters. The computer will favor the paths on the search tree that lead to the highest valued board configurations.

F. MINIMAX

When the computer is evaluating possible paths, the feasibility of a path is dependent on the opponent making certain moves. The absolute best path for the computer will necessarily require its opponent to make the worst possible moves for himself and, conversely, the best possible moves for the computer. This is an ostensibly unrealistic assumption. In order to address this problem, the computer takes advantage of a concept known as a MINIMAX search method. This method assumes that an opponent will make moves that are desirable for him and not the computer. The program will usually calculate the best move for the opponent and assume he will take it, thus limiting the number of branches with which to work. The computer will choose, from among the remaining paths, that which leads to the best board configuration at the bottom level of the tree.

G. Coding Rhythmomachy

Rhythmomachy is a very interesting and unique game from a coding perspective. There are several aspects of the game that contribute to its complexity.

In Rhythmomachy it is only necessary that a player setup a capture. This opens the possibility of a player taking more than one piece at one time. The idea that a piece need only be endangered added to the complexity of the evaluation function.

Unlike many other games, a capture in Rhythmomachy does not remove the piece from the board. Instead, it becomes one of the capturing player's pieces. This unusual rule creates an added challenge when writing the code for the game tree. Since no piece can be removed from the board, the search tree's branching factor has the potential to increase well beyond the average from turn to turn. This is again different from chess where pieces are always removed from the board, causing the branching factor to gradually decrease.

H. Parallel Processing on the T3E

The Cray T3E supercomputer is currently ranked the twenty-third fastest computer in the world. The Cray's speed does not come from just one processor, however. The T3E gains its speed from 512 450-megahertz networked processors. By using parallel processing techniques, this computer is able to do calculations several orders of magnitude faster than any desktop computer could.

The nature of the Cray's construction makes writing programs for it a process which is completely different and much more complex than coding on a standard serial machine. Programs for the Cray must distribute all of their tasks among the many processors so that each processor can handle a small piece of the computation. The Cray's speed advantage stems directly from this division of labor.

To parallelize the game tree for use on the CrayT3E, the Parallel Virtual Machine (PVM) library was utilized. The PVM library includes several functions and commands that allow programs to be adapted to run on multiple processors. PVM was developed at Oak Ridge National Laboratory as a way to use networked computers as one large parallel machine. It is used by researchers in a variety of scientific computing applications and by instructors to educate students in the art of parallel programming. One of the most important features of PVM is its ability to pass messages between processors. The Rhythmomachy game player takes advantage of this capability by spreading the computation of the game tree among several processors.

III. Procedure

A. Data Structures

The game engine is based fundamentally on two primary data structures. The structure "Square" contains all the information that a single location on the board needs to know. Each space knows whether or not a piece is resting on it, and the color, shape, and number of that piece. (Detailed descriptions of the member functions of "Square" appear in the appendix.)

The Structure “Board” contains all the information about the state of a game at a particular moment. The most important piece of information contained in “Board” is a two dimensional array of Squares that represents the arrangement of the pieces on the board. In order to simplify other aspects of the program, the Board structure has been made responsible for moving pieces, detecting captures, performing captures, and keeping track of the current score. This structure also allows data to be saved for an unfinished game.

At the current time, there exists a fifteen minute interactive run-time limit on the Cray imposed by the Pittsburgh Supercomputing Center. However, a game of Rhythmomachy can take much longer. This problem is addressed by the save game feature in the game. The program saves the current board after each move by the human player. The format of the saved game files is described in Table 1.

Table 1: Saved Game Coding Format

Line #	Possible Values	Meaning
1	0 or 1	0 for Blacks turn, 1 for Whites turn
2	Integers	Points for black
3	Integers	Points for white
4	Integers	Black’s captures
5	Integers	White’s captures
6	Integers	Point total of black’s captures
7	Integers	Point total of white’s captures
8-72	-1, 0, 1; -1, 0, 1; integers	color, shape, number of 64 squares on the board

Other data structures are used for convenience throughout the code such as a structure named “Pos” used to store a row and a column, and a class named “Priorities” used to define piece placement after captures.

B. The Capture Function

In each situation, to make a capture, three conditions must be satisfied. First an enemy piece must be located at the appropriate distance. Secondly, a second friendly piece must be present within an appropriate radius of the enemy piece. Finally, the function must determine whether or not the numbers on the two friendly pieces are arithmetically compatible with the enemy piece. If each of these conditions are met and a capture is detected, then the capture is performed.

- A circle is moved and performs a capture with another circle
- A circle is moved and performs a capture with a square.
- A square is moved and performs a capture with a circle.
- A square is moved and performs a capture with another square.

The actual execution of captures is carried out by the “moveSquare” function. Once the capture is enacted, the captured piece is placed in the corner of the board corresponding to the new owner of the piece. The captured piece’s new location is determined by a class created for just that purpose. This class stores a set of priorities for the placement of pieces in an array. When a piece is captured, this array is referenced from the highest priority down, until a space is found which is open and appropriate for placement.

C. Player vs. Player (PVP)

The player versus player portion of our computer program was the first to include a user interface. This portion of our program was designed to display the game board, graphically change the location of the pieces on the board, allow user input, and address any inappropriate input by the user.

The display board portion of the code is a series of for loops and if statements that construct the board out of a sequence of letters and symbols. This code was optimized for speed as well as readability. Our team felt that, by emphasizing these two factors in the display board function, the game would become much more enjoyable.

D. The Search Trees

To design the program’s decision-making facility, we began by implementing a rudimentary serial one-level search tree. This was accomplished by first writing a routine that allowed the program to “visualize” board states after all possible moves and then adding the functionality of heuristic evaluation which had been produced in a separate routine. In a one level tree, the computer can only see one move ahead, and thus cannot predict any responses by the opponent. This limited insight makes the one level program relatively easy to defeat.

After the one-level tree was compiled and run successfully, the tree was expanded to have a greater “look ahead” capability, allowing it to evaluate more board positions before finalizing its decision. This process was slightly more complicated, because a larger tree uses MINIMAX, and therefore needs to function in different ways at different levels. Because of this, we had to write unique functions for each level of the tree. We finally completed a four-level search tree. It ran, but the limitations of serial processing meant that a single search could take as much as five minutes. We then turned to the power of parallel supercomputing to optimize the search and ensure a faster decision-making process.

E. The Evaluation Function

Because it is impractical for a computer to consider the entire game tree for Rhythmomachy, it becomes necessary to utilize a heuristic evaluation function. We considered many parameters for the function before finally settling on four important ones. The most heavily weighted factor is the number of pieces captured. Second is the number of points earned. Finally, the program considers the number of enemy pieces endangered, and of least priority, the number of friendly pieces in positions near the center of the board.

The first two conditions are important because they represent progress toward eventual victory in the game. The third condition takes on importance because the opponent's moves are somewhat limited by which of his pieces are endangered. Finally, the fourth condition is somewhat important because, as in chess, pieces in the center of the board command a great deal more authority than do pieces around the edges.

Another important aspect of our evaluation function is its win-checker. Before the heuristic function evaluates the above four conditions and assigns relative values appropriately, it first checks to see whether either side has won the game. If this is the case, it breaks immediately and returns a value of positive or negative 32000, depending on which side has attained the win. This is vital, because our MINIMAX technique dictates that a branch in which the game can be won is almost certainly a branch in which the game will be won, assuming each player makes the perfect move on any given turn.

Since many different branches of the tree could include wins, there had to be a way to assign priority to some over others — it is more desirable to head toward a win in a closer level of the search tree. However, two different winning boards would receive the same evaluation at different levels under the standard rules of MINIMAX. In order create a preference for more immediate computer wins, the evaluation function subtracts the level on which a win occurs from the value returned. Similarly, to create an aversion toward more immediate opponent wins, the function adds the level of the board to the value returned.

F. Parallelizing the Code

To parallelize the search tree, we designed a system wherein PE 0 runs the initial visualization function and sends each first-level board configuration to another processor. The other processors then continue visualizing new boards until they reach the fourth level of the tree, at which point MINIMAX transmits each processor's results back to PE 0 for assembly and evaluation. PE 0 then carries out the chosen move using the same routines that we had written for the serial search tree. This message passing process is illustrated for eight processors in Figure 2.

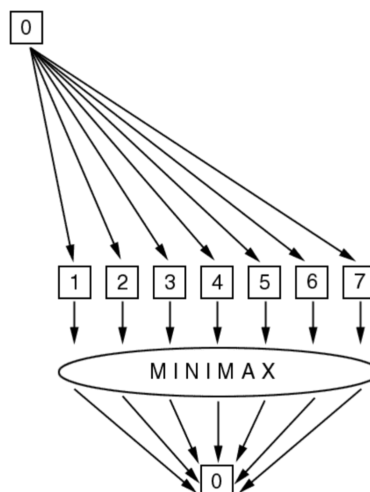


Figure 2: The Message Passing Process

G. The Tournament

A human Rhythmomachy tournament was devised in order to analyze and observe player strategies as well as to create an experienced player pool. The tournament produced a group of experienced players against which the search tree could be tested. This would allow our team to see how the program fared against a variety of opponents each with varied skill levels and strategies. The tournament also allowed our team to discover any problems inherent in the rules and setup of the game and to modify the rules if necessary.

The tournament utilized a standard binary tree structure. We began with 32 players, each of whom played a game against one opponent. The winner of each of these first 16 games advanced to the second round. The second round winners advanced again. This pattern continued until the final game. The winner of this game was declared the overall champion of Rhythmomachy within the PGSS. The scores were recorded from each game and the data was compiled and analyzed in the hopes of finding new strategies that could be implemented in the computer program.

H. Testing the Game Against People

In order to test our program, we pitted it against several human opponents. We grouped the pool of players into three classes based on the tournament. First there was a group that had no experience playing the game. In the middle was a group who had played the game, but had no wins against a human opponent. Finally, the members of the last group had each won at least one game in the tournament. Each of these sample groups played against two different versions of the game. The first version implemented a one level search tree, and the second trial was against our parallelized four level search tree.

I. Computer vs. Computer Testing

To improve and refine the skills and strategy of our program, we created a few different evaluation strategies and tested the original evaluation function against them. First, we pitted the original against itself to find out whether white or black had an advantage. After confirming that neither side came out on top consistently, we initiated a round-robin style tournament using three different strategies.

First, the original did battle with an opponent who had an affinity to capture square zeros and square twos. (We discovered that this was an effective strategy that humans had used against the computer in the computer vs. player testing.) We then generalized this strategy to create an opponent that favored all squares. Similarly to its predecessor, this strategy moved toward board configurations in which it possessed more squares. To further test the effectiveness of the original evaluation function, we also created a much more aggressive strategy. This strategy placed a greater weight upon capturing the opponent's pieces and less on defending its own.

IV. Results

Our game-playing program functioned properly both serially and when played in parallel on the Cray. We were able to run it in a timely fashion with at most a three level search tree on a serial computer, and

up to four levels with parallel processing. The parallel version of the program calculated approximately 40,000,000 nodes in four seconds on four processors.

When tested against a diverse pool of human opponents, the one level tree fared reasonably well. When it played 15 human players, the level one program won 80% of the time. The four level tree fared predictably much better than the one level tree. It boasts a near perfect success rate against human opponents. This version of the search tree won 92% of the 66 games that it played. The data from our human versus computer trials is shown in Figures 3 and 4.

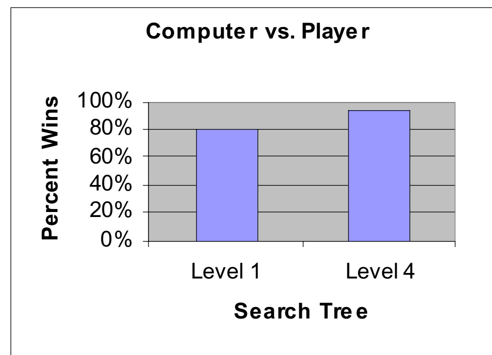


Figure 3: Program Success Rate Against Human Opponents

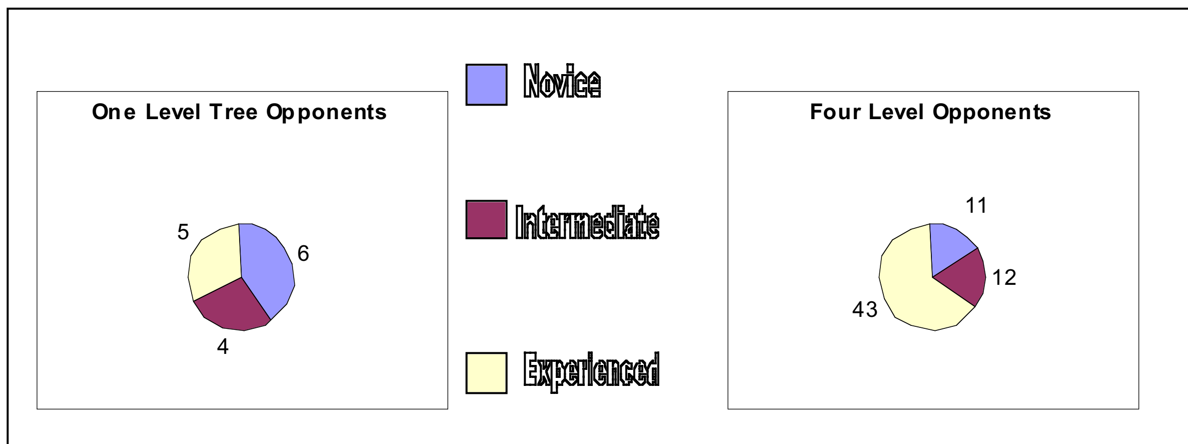


Figure 4: Player Skill Level Breakdown for Player vs. Computer

The three new evaluation functions involved in the computer vs. computer tournament were named Squares02, AllSquares, and Aggressive. The original evaluation function was known as Sledge. The results of the games among these four functions are shown in Table 2 on the following page.

Table 2: Computer vs. Computer Data

Between		Percent Wins of Strategies Below			
Strategies		Sledge Wins	Squares02 Wins	AllSquares Wins	Aggressive Wins
Against	Sledge	X	70%	40%	0%
	Square02	30%	X	50%	X
	All Squares	60%	50%	X	X
	Aggressive	100%	X	X	X

V. Discussion

A. Strategy

Our game was set up in such a way that the level of searches performed by the computer could be easily altered. As the level of search increases, the computer becomes much more difficult to defeat. As a player it is advantageous to develop a strategy in order to defeat an opponent. However, when one's opponent thinks four moves ahead, this can be a daunting task. Our team was faced with this problem when we found ourselves constantly losing to the four level tree. We tried in vain to develop a efficacious strategy.

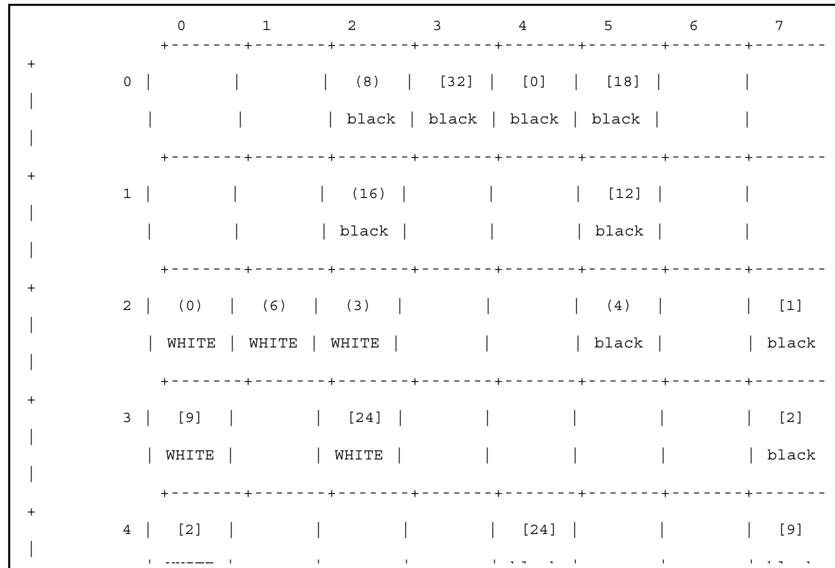
While we were never able to develop an effective method to reliably defeat the four level tree, we did discover a strategy for defeating the one level tree. This strategy entails allowing the computer to make the first offensive move by making a defensive opening move. The reasoning behind this strategy lays in the nature of a one level tree. With a one level search tree it is impossible for the computer to look ahead and act defensively, and so by always acting on the computer's move, we were able to look at least one move further than the computer could. In practice, this strategy was very effective.

B. The One Level Tree Versus the Four Level Tree

As could be expected, the four-level search tree performed significantly better than the one-level tree. The data shows this in two ways:

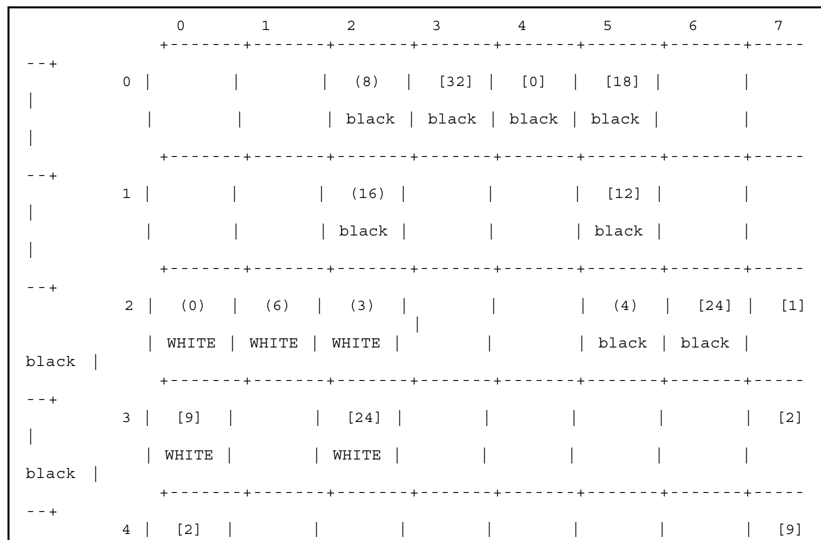
First, players have a better win-loss record against the one-level tree than against the four-level tree. Several players have won games against the one-level tree, but humans have almost never shown the ability to defeat the program running a four-level tree. This indicates a higher proficiency level and aptitude for winning games on the part of the four-level tree.

Secondly, game transcripts clearly show that the four-level tree plays in a more strategic manner. The following diagrams demonstrate this. Figures 4 and 5 show the board states before and after a move chosen by a one-level search tree respectively. By determining the strategic value of the position attained through this move, we see that it is significantly less desirable than that acquired through the four-level tree's response to the same initial situation, shown in Figures 6 and 7.



	0	1	2	3	4	5	6	7
0			(8) black	[32] black	[0] black	[18] black		
1			(16) black			[12] black		
2	(0) WHITE	(6) WHITE	(3) WHITE			(4) black		[1] black
3	[9] WHITE		[24] WHITE					[2] black
4	[2]				[24]			[9]

Figure 4: One Level Tree Before Movement



	0	1	2	3	4	5	6	7
0			(8) black	[32] black	[0] black	[18] black		
1			(16) black			[12] black		
2	(0) WHITE	(6) WHITE	(3) WHITE			(4) black	[24] black	[1] black
3	[9] WHITE		[24] WHITE					[2] black
4	[2]							[9]

Figure 5: One Level Tree After Movement

The program chose to move its [24] from (2,6) to (4,4). Clearly it was thinking about capturing its opponent's [24] using its [24] and [0] – in itself a prudent action to take. However, its implementation of the strategy is flawed. By moving its [24] to (4,4), it enables the player to capture the [24] with at least two different methods: addition with White's [24] and [0], and multiplication with White's [12] and [2]. Thus, the one-level search tree's performance in this situation is less than optimal. This error is to

be expected, however, as the program has no capability to look ahead in the interest of protecting its own pieces.

```
          0       1       2       3       4       5       6       7
              +-------+-------+-------+-------+-------+-------+-------+-----
--+
          0 |       |       |  (8)  |  [32] |  [0]  |  [18] |       |
|
            |       |       | black | black | black | black |       |
|
              +-------+-------+-------+-------+-------+-------+-------+-----
--+
          1 |       |       |  (16) |       |       |  [12] |       |
|
            |       |       | black |       |       | black |       |
|
              +-------+-------+-------+-------+-------+-------+-------+-----
--+
          2 |  (0)  |  (6)  |  (3)  |       |       |  (4)  |  [24] |  [1]
|
            | WHITE | WHITE | WHITE |       |       | black | black |
black  |
              +-------+-------+-------+-------+-------+-------+-------+-----
--+
          3 |  [9]  |       |  [24] |       |       |       |       |  [2]
|
            | WHITE |       | WHITE |       |       |       |       |
black  |
              +-------+-------+-------+-------+-------+-------+-------+-----
--+
          4 |  [2]  |       |       |       |       |       |       |  [9]
|
```

Figure 6: Four Level Board Before Movement

```
          0       1       2       3       4       5       6       7
      +-------+-------+-------+-------+-------+-------+-------+-------+
    0 |       |       |  (8)  |  [32] |  [0]  |  [18] |       |       |
      |       |       | black | black | black | black |       |       |
      +-------+-------+-------+-------+-------+-------+-------+-------+
    1 |       |       |  (16) |       |  [24] |  [12] |       |       |
      |       |       | black |       | black | black |       |       |
      +-------+-------+-------+-------+-------+-------+-------+-------+
    2 |  (0)  |  (6)  |  (3)  |       |       |  (4)  |       |  [1]  |
      | WHITE | WHITE | WHITE |       |       | black |       | black |
      +-------+-------+-------+-------+-------+-------+-------+-------+
    3 |  [9]  |       |  [24] |       |       |       |       |  [2]  |
      | WHITE |       | WHITE |       |       |       |       | black |
      +-------+-------+-------+-------+-------+-------+-------+-------+
    4 |  [2]  |       |       |       |       |       |       |  [9]  |
      | WHITE |       |       |       |       |       |       | black |
      +-------+-------+-------+-------+-------+-------+-------+-------+
    5 |  [1]  |       |  (4)  |       |       |  (3)  |  (6)  |  (0)  |
      | WHITE |       | WHITE |       |       | black | black | black |
      +-------+-------+-------+-------+-------+-------+-------+-------+
    6 |       |       |  [12] |       |       |  (16) |       |       |
      |       |       | WHITE |       |       | WHITE |       |       |
      +-------+-------+-------+-------+-------+-------+-------+-------+
    7 |       |       |  [18] |  [0]  |  [32] |  (8)  |       |       |
      |       |       | WHITE | WHITE | WHITE | WHITE |       |       |
      +-------+-------+-------+-------+-------+-------+-------+-------+
    You: 0 = 0 points.   Me: 0 = 0 points.
```

Figure 7: Four Level Board After Movement

The four-level tree's response, demonstrated in Figures 6 and 7, is much more impressive. Although it utilizes the same strategy (i.e., setting up one piece to capture the white [24]), it does so in a manner that does not endanger its own [24]. Its increased "look ahead" capacity has been a great benefit, enabling it to play more soundly in the long run.

C. Computer vs. Computer Testing

Our goal to create an intelligent game-playing algorithm could not be reached simply creating a working game tree. The intelligence of the algorithm relies heavily on the "wisdom" of the function that evaluates the state of the game. Originally, the relative value of the pieces was never taken into account. This fact allows an insightful opponent to exploit the program's weakness. Because our first evaluation function was limited to seeing only four moves ahead, it could not accurately weigh the long-term relative value of pieces. By pitting it against a variety of adversaries, however, we were able to analyze its failings and refine its strategy.

VI. Acknowledgements

The members of PIG-PAR would like to thank, in no particular order:

PGSS Director Dr. Peter Berget, for making the program run smoothly and giving us those extra ten minutes during the Symposium.

Team Project Director "Captain" Kirk Yenerall, for all of his support, dedication, and enthusiasm toward each of the Computer Science projects.

Teaching Assistant Matt Huenerfauth, for putting up with us 15 hours a day for almost an entire week… even when we started mumbling gibberish and threatening to mutiny.

The Pittsburgh Supercomputing Center, for providing us with Cray time and trusting us not to sell it to foreign dictators of questionable moral fiber.

Certified genius Richard Dore, for taking several days out of his "busy" schedule to rage against our machine.

`yaoj`, a.k.a. "Jixian Yao," for constantly taking up 360 processors just to analyze the "Hello, world" function. Drop him a line and thank him for us: jxyao@orangutan.stanford.edu.

Sledge, for being the cutest and most lovable artificially intelligent parallel Rhythmomachy player that any team of programmers could ever hope to ask for.

The guys at the O, for always being there when we just needed someone to talk to.

VII. Works Consulted

16th Century Rhythmomachy, Version 1, "http://www.inmet.com/~justin/game-recon-rhyth1.html," 27 July 1999.

Carl Burch, Surveying the Field of Computing: Third Edition, 1999.

Cray T3E White Papers: Performance of the Cray T3E Multiprocessor, "http://www.sgi.com/t3e/performance.html," 27 July 1999.

Deitel and Deitel, C++: How to Program. 1994. Prentice Hall, Englewood Cliffs, NJ. p. 653

Gallagher, Huenerfauth, Kimbrell, Maust, Miller, Ramsey, Yoon, Yu, "Supercomputing-Parallel Algorithms for Multiplexor: An Investigation of Artificial Intelligence and Game Theory", Journal of the Pennsylvania Governor's School for the Sciences, Pennsylvania Governor's School for the Sciences and Carnegie Mellon University, Volume 15, 1997.

Jackson, Philip C. Introduction to Artificial Intelligence. 1985. Dover Publications, New York. pp. 123-138.

The Philosopher's Game "http://www.inmet.com/~justin/fulke.html," 27 July 1999.

PVM: Parallel Virtual Machine, "http://www.epm.ornl.gov/pvm/pvm_home.html," 27 July 1999.

Luger, George F., Stubblefield, William A. Artificial Intelligence and the Design of Expert Systems. 1989. The Benamin/Cummings Publishing Company, Inc., New York. pp. 35-38, 150-153, 170-178.

Appendix A: Description of Functions and Syntax

Table 3: Member Functions of Class "Board"

Function Name	*Function Use: Possible Values Returned (if applicable)*
`InitBoard()`	Reads initial board configuration From the file initboard.txt
`SaveBoard(char*)`	Saves board configuration to a file
`LoadBoard(char*)`	Reads board configuration from a file
`GetTurn()`	Returns whose turn it is: BLACK or WHITE
`GetBlackPoints()`	Returns BLACK's points: integers
`GetWhitePoints()`	Returns WHITE's points: integers
`GetBlackCaptureCount()`	Returns BLACK's number of captures: Integers
`GetBlackCaptureTotal()`	Returns BLACK's points from captures: Integers
`GetWhiteCaptureCount()`	Returns WHITE's number of captures: Integers
`GetWhiteCaptureTotal()`	Returns WHITE's points from captures: Integers
`GetShape(int rows, int columns)`	Returns shape of piece at (r,c): EMPTY, CIRCLE, SQUARE or OFF_BOARD
`GetColor(int rows, int columns);`	Returns color of piece at (r,c): EMPTY, CIRCLE, SQUARE or OFF_BOARD
`GetNumber(int rows, int columns);`	Returns number on piece at (r,c): Integers, EMPTY

Table 3: Member Functions of Class "Board" (cont'd)

`SetColor(int row,int column,int MyColor)`	Sets color at (r,c) to myColor: BLACK, WHITE
`SetNumber(int row,int column,int MyNumber)`	Sets number at (r,c) to myNumber: integers
`SetShape(int row, int column,int MyShape)`	Sets shape at (r,c) to myShape: CIRCLE, SQUARE
`SetTurn(int color)`	Sets turn to color, BLACK or WHITE
`SetBlackPoints(int points)`	Sets black points to points: integers
`SetWhitePoints(int points)`	Sets white points to points: integers
`SetBlackCaptureCount(int count)`	Sets black capture count to count: integers
`SetBlackCaptureTotal(int total)`	Sets black capture total to total: integers
`SetWhiteCaptureCount(int count)`	Sets white capture count to count: integers
`SetWhiteCaptureTotal(int total)`	Sets white capture total to total: integers
`MoveSquare (int sr, int sc, int fr, int fc, bool displayOp=false);`	Moves a square from (sr,sc) to (fr,fc): 1 if successful and 0 otherwise.
`IsCaptures(int row, int col, bool displayOp=false)`	Detects capture and returns the Position of the capturable piece

Table 4: Member Functions of Class "Square"

Function Name	*Function Effect*	*Possible Values Returned*
`GetShape()`	Returns the shape of the piece,	CIRCLE, SQUARE, or EMPTY
`GetColor()`	Returns the color of the piece,	BLACK, WHITE, or EMPTY
`GetNumber()`	Returns number of piece	Integer or EMPTY
`IsOccupied()`	Returns occupation status	True or false
`MakeEmpty()`	Sets all of the attributes of the square to EMPTY	EMPTY
`SetColor(int myColor)`	Sets the color to myColor:	BLACK, WHITE, EMPTY
`SetNumber(intmyNumber)`	Sets number to myNumber, EMPTY	Integer or EMPTY
`SetShape(int myShape)`	Sets shape to myShape:	SQUARE, CIRCLE, EMPTY

Appendix B: Full Data File from Computer vs. Player/Computer vs. Computer Tournaments

```
Rhythmomachy Team Project Data File

Class 1 = Experienced Players from Tournament
Class 2 = Intermediate Players from Tournament
Class 3 = Novice Players from Tournament

Level4
------
W/L   Human Comp. Class
Loss  -     -     1
Loss  -     -     1
```

Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	-	-	1
Loss	3/5	6/25	1
Loss	0/0	4/32	1
Loss	3/21	4/33	1
Loss	3/19	4/34	1
Win	4/26	0/0	1
Loss	2/30	4/28	1
Win	7/34	2/14	1
Loss	5/53	2/14	1
Win	4/32	3/15	1
Win	4/50	2/8	1
Win	4/26	0/0	1
Loss	2/30	4/33	1
Loss	2/6	4/51	2
Loss	2/26	4/28	2
Loss	2/12	4/33	2
Loss	1/6	4/44	3
Loss	2/26	4/27	2
Loss	1/3	4/44	3
Loss	1/0	4/26	1
Loss	1/8	4/41	1
Loss	0/0	4/39	1
Loss	0/0	6/27	2
Loss	0/0	4/34	2
Loss	1/24	4/35	2
Loss	1/8	4/35	3
Loss	2/9	5/46	3
Loss	1/0	5/46	2
Loss	1/6	4/62	2
Loss	1/0	4/68	1
Loss	1/12	4/53	3
Loss	0/0	4/35	3
Loss	1/2	4/27	3
Loss	0/0	4/30	1
Loss	0/0	4/84	1
Loss	1/12	4/38	1
Loss	1/12	4/33	2

```
Loss  0/0   4/25  3
Loss  3/20  4/50  3
Loss  1/6   4/38  3
Loss  1/18  4/44  3
Loss  2/9   4/30  2
Loss  0/0   4/34  2

66 Games - 5 Human Wins - 61 Human Losses

43 - Experienced players
12 - Intermediate players
11 - Novice players

Level1
------
W/L   Human Comp. Class
Loss  0/0   4/37  3
Loss  3/35  4/26  3
Loss  3/36  5/28  3
Loss  0/0   4/60  -
Win   -     -     1
Win   -     -     3
Loss  -     -     2
Loss  -     -     3
Loss  -     -     2
Loss  -     -     1
Loss  -     -     2
Loss  -     -     1
Loss  -     -     1
Win   -     -     1
Loss  3/3   4/41  1

15 Games - 12 Human Losses - 3 Human Wins

Sledge v Squares02
-----------
7/3

Sledge v AllSquares
----------------
6/4

Squares02 v AllSquares
------------
5/5
```

Appendix C: Program Code

The code for this team project may be found on the compact disk which accompanies this Journal.