An Artificially Intelligent Battleship Player Utilizing Adaptive Firing and Placement Strategies

Jeremy G. Bridon, Zachary A. Correll, Craig R. Dubler, Zachary K. Gotsch

The Pennsylvania State University, State College, PA 16802, USA

The purpose of this paper is to provide a plan for an Artificially Intelligent (AI) player for the classic game of Battleship. In order to accomplish this goal, the game is split into three logical areas: ship placement, targeting, and sinking strategy. This paper details the specific AI objectives for each area, as well as the implementation of an intelligent system using genetic algorithms. This artificially intelligent player will be tested in a competition against other artificially intelligent battleship opponents.

I. Introduction

THE CLASSIC game of Battleship involves two players, each placing five ships of lengths two, three, three, four, and five onto their ten by ten unit board. Taking turns, they select board positions to strike. After an entire ship has been struck by an opponent it is "sunk" and removed from the board, and the player who sunk the ship is notified which type of ship was removed. Once all of a player's ships have been removed, the game is over and won by the player who still has ships remaining.

The use of heuristic algorithms, algorithms that provide acceptable solutions but lack a formal proof, are become much more commonplace in solving both dynamic goal and computationally intense problems [4]. They are commonly used to search for solutions to NP-complete problems and are a popular topic in Artificial Intelligence research. Since the game involves another player whose behavior is unknown, no formal algorithmic solution exists. However, a heuristic algorithm can find arising patterns in the opponent's play and potentially solve, or win, a game more quickly than naive, hard-coded, placement.

The proposed solution uses genetic algorithms, a form of heuristic algorithm, which is inspired by the evolutionary paradigm to create an approach that is tailored to the opponent's strategy. By observing the opponent's placement and targeting strategies over multiple games, the solution adapts its strategy to counter the opponent's, which leads ships to be placed in positions that have been given little attention and to fire more often at positions which have commonly held the opponent's ships.

II. Methodology

In the field of Artificial Intelligence there are three possible approaches that are well suited to this problem definition: rule-based expert systems, genetic algorithms, and neural networks. All three of these areas of AI can effectively play a game of Battleship, however, the solution presented in this paper utilizes genetic algorithms in its approach to Battleship. Each approach was thoroughly investigated by the authors, with observations on each method's strengths and weaknesses discussed below.

Rule-based expert systems use user-created "rules" to govern the behavior of the algorithm. These rules are dependent on the knowledge of the expert who is providing insight into the game. Since none of the authors of this paper are experts in Battleship, this approach was discarded in favor for an approach that does not depend on the authors' knowledge of the game. More importantly, a rule-based expert system cannot learn at run-time, though it can make correct deductions based on the rules that it is given. These are not dynamic enough to adapt to the opponent's strategies. Rule-based systems may be efficient for sinking logic as the logic can be quickly hard-coded with the system optimizing certain steps. However, other research has proved that such a system can still be outperformed by code generated by a genetic algorithm [8].

An approach utilizing neural networks was discarded due to the "black box" characteristics of the algorithm, that is, the inability to see the inner workings of the system. Though the system is fundamentally much more dynamic than a rule-based expert system, a solution for ship placement and shooting is very complex compared to a hard-coded statistical analysis [9]. Such a system also lacks the ability to easily represent linear logical instructions, which are needed for ship sinking.

To be competitive a solution needs to be able to find enemy ship placement patterns in order to raise the likelihood of hitting an enemy ship. Once a ship has been hit, logic needs to be applied to sink the target. Though this could be hard-coded, there is proof [8] that heuristic methods may be more effective than conventional, natural methods. In order to stay in each game as long as possible, the solution must also find enemy targeting patterns and attempt to avoid their own learning systems. To remain competitive the solution needs to adapt quickly enough to win even in small rounds of eleven or fewer games. Genetic algorithms can be a powerful method for finding quick and dynamic solutions at run-time [4]. Solution representation is dynamic enough to represent the several components of this problem: Ship placement, shooting, and sinking.

According to [3,6,7], Monte Carlo Methods are helpful when working with non-linear probability densities. One method explains how to apply Monte Carlo Methods to a two dimensional grid search when localizing a robot within a given map [3]. This is helpful for both ship placement and shooting since a parallel between pastern matching in robot localization and ship placement can be drawn. Ship sinking is complex as

it needs run-time logic that can be changed for shot optimization. Genetic programming is a valid method of instruction representation and optimization for linear logical instructions [1].

III. AI Implementation

The game of Battleship can be split into three distinct areas: placement of ships, ship targeting, and sinking logic. Ship placement uses an adaptive statistical method, ship targeting utilizes a genetic algorithm to find and analyze patterns, and genetic programming is used to create and optimize the ship-sinking logic.

Pre-training the solution is important since an initial "random" solution might perform too poorly, giving few hits on the opponent's ships and little data for training the targeting algorithm. Each new opponent resets the internal data to this pre-trained solution set. The pre-training is done by playing this solution design against several hard-coded opponents. After each round with the same opponent, the solution will attempt to optimize for the current opponent's method of play. If games are to be played against the same opponent in the future, the optimized solution against that player could easily be saved and restored.

A. *Ship Placement*

The solution to ship placement is an adaptive algorithm that considers statistical data as well as the opponent's shot placement over time. This algorithm uses the number of different ways a ship can be placed in an individual cell as its base information. The algorithm then uses a Sequential Monte Carlo Method to place ships in the probabilistically least likely location to be shot, but not exclusively [6]. The strength of this algorithm is in the Sequential Monte Carlo Method which allows for adaptation at run-time.

After randomly placing ships within a grid of size 10x10 for 1,000,000 iterations, the number of times a ship is placed in a certain cell is recorded. Figure 1 shows the sum of the five ships' probabilities. The graph shows that the randomly placed ships are symmetric about the center of the grid but are not completely Gaussian. The inverse of the sum of the grids is taken and then normalized in order to have locations where the ship is less likely to be placed with a higher weight.

A Sequential Monte Carlo Method is implemented in order to place the ships in the statistically least likely places on the grid. Monte Carlo Methods use grid-based Markov methods in order to deal with the multi-modal, non-Gaussian, nonlinear densities[8]. Monte Carlo Methods have been used since the 1950s to generate weighted random values from given probability distributions in areas from robot localization to financial market prediction [6].

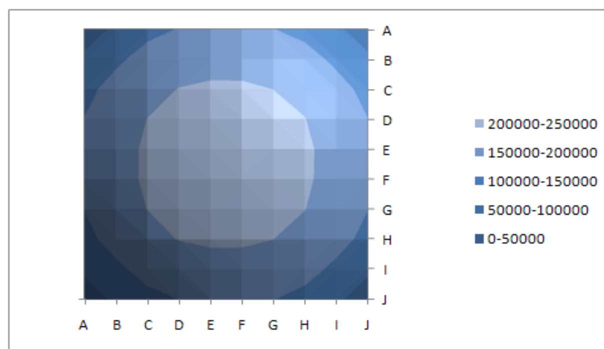
While the placement of ships for the first game is hard coded and based on prior statistical information, the algorithm displays intelligence by adapting to the opponent over time. The algorithm logs the opponent's shot selections and decreases the weights of the cells that were targeted during the previous games when placing the ships. For each game following the first one, the prior statistical data as well as newly acquired information about the opponent's shot locations will be utilized to place the ships.

A shortfall of Monte Carlo Methods is their computational intensity [7]. While ships could be placed in a purely random fashion, Figure 1 displays that their placement will tend towards the center of the grid. A smart opponent will notice this pattern and place more shots around the center than the corners. By introducing a Monte Carlo sampling method, the center is still the most likely place a ship will be placed, however other areas such as the corners will have a higher likelihood than random placement.

B. *Targeting and Shooting*

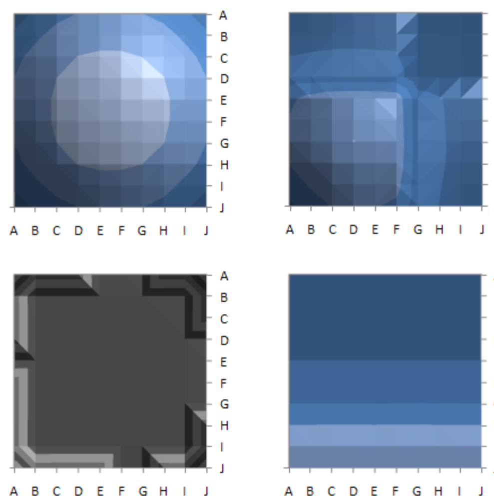
The core of solving the classic game of Battleship is adaptive and efficient targeting and shooting. Shot placement needs to take advantage of the opponent's ship placement strategies; however, in order to make observations about the opponent's placements and discover trends, an efficient way to represent the placement data is needed.

FIGURE 1.



Random ship placement densities over 1,000,000 rounds.

FIGURE 2.



Different methods of ship placement over 1,000,000 rounds. From top left, clockwise: Random, lower-left corner favored, bottom wall, one ship in each corner.

Figure 1 demonstrates the random placement densities of all five ships on a 10x10 board for over 1,000,000 games. A pattern emerges in which the center experiences many more placements since edge squares are not filled by as many ship placement combinations. Different, but clearly visible, patterns emerge with different placement algorithms, as seen in Figures 1, 2, 3, which suggests a pattern-based solution.

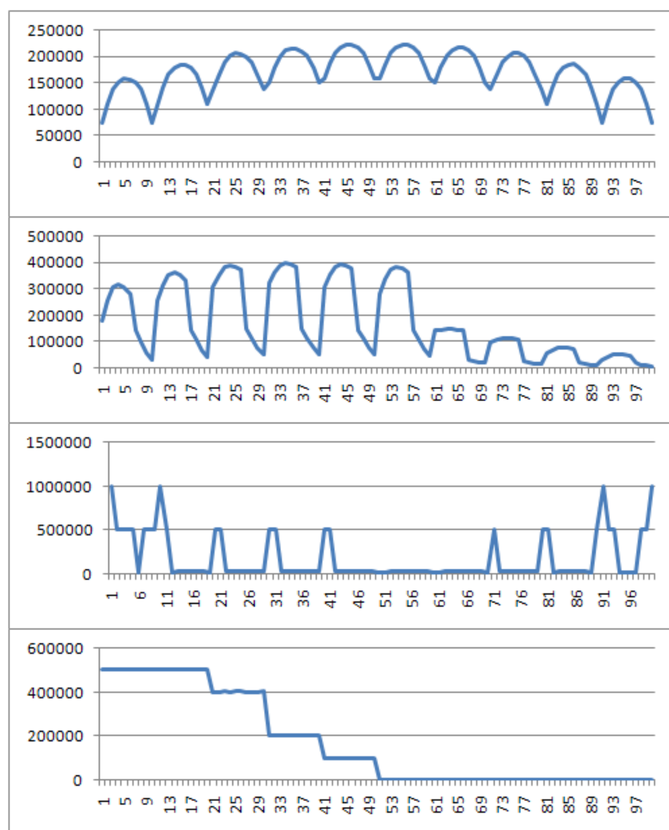
Different methods of ship placement over 1,000,000 rounds. From top left, clockwise: Random, lower-left corner favored, bottom wall, one ship in each corner.

The reader can observe several useful facts from a chart of this type, most importantly trends in the opponent's placements. Figure 2 is a chart of several advanced placement strategies as follows:

- Place all ships starting in the bottom row and extending upwards
- Place a ship in each corner and the final ship randomly
- Place all ships in the lower left quadrant with high probability

However, while this square histogram format is very easy for a human to understand, it is not easy to generate a three-dimensional function that can approximate the described densities. To simplify the problem, the data is concatenated into a continuous one-dimensional array. The graphs shown in Figure 2 are shown in the linear format in Figure 3.

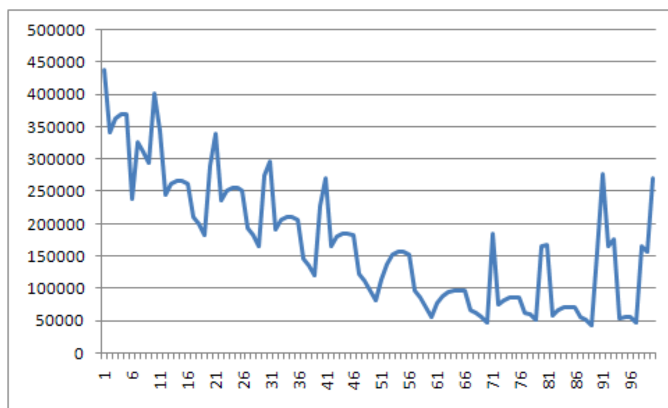
FIGURE 3.



Different methods of ship placement over 1,000,000 rounds. From top to bottom: Random, lower-left corner favored, one ship in each corner, bottom wall.

Another observation made is that density graphs representing placement strategies in this way all resemble a harmonic wave. Therefore, if a harmonic approximation can be fitted to this curve, the solution can use the approximation as a representation of the probability that a ship will appear in any given square. This approximation will also allow the probabilities to be more regularly distributed among the squares, which is helpful when adapting to new strategies. The representation of data in this way also allows us to create a composition of multiple strategies, as well as prepare for expected phenomena such as random placement strategy. Figure 4 is a graph of a combination of the three strategies featured in the previous graph as well as the random pattern strategy.

FIGURE 4.



Combination of strategies and random placement.

Since all harmonic functions can be decomposed into a combination of sine and cosine waves, the solution utilizes a genetic algorithm to fit a series of waves to a probability function. Each combination of waves is represented as a gene, and by using crossover and mutation, they converge to the probability distribution that has been constructed both from data gained in previous games with the same opponent and "training" data which is entered before the match begins. Each gene is made up of five harmonics, with each having four parameters. The harmonics are represented by the formula

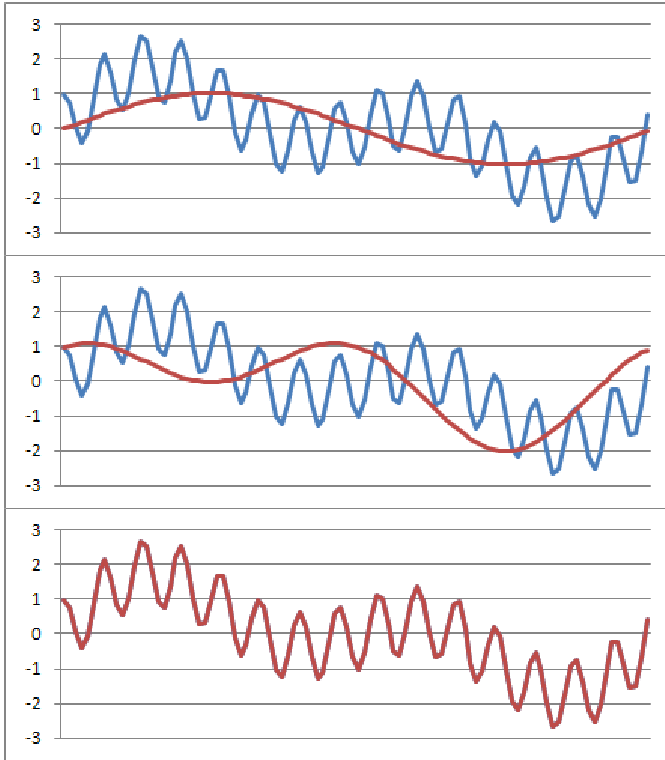
$$\alpha \sin(\mu t) + \beta \cos(\omega t)$$

Figure 6 contains a sample gene of only two harmonics that solve for random ship placement. The wave represented by the gene is the summation of these five harmonics. A genetic algorithm is composed of five basic steps: initialization, selection, reproduction, termination, and conclusion.

The initialization step is the creation of the initial population. In this algorithm the population is maintained at 100 members. When playing with a new opponent, eighty percent of the initial population is randomly generated, while the other twenty members are pre-generated at intervals spanning the problem space. In the case of a game with the same opponent, the population is carried over from the previous game or loaded from a saved gene pool.

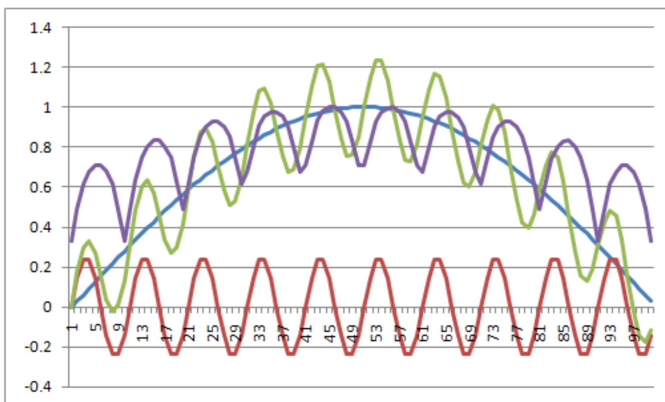
The selection step chooses which members of the population will reproduce and pass their traits on to the next generation. A fitness score determines how successful or poor a gene is, that is, whether it should be allowed to reproduce. The fitness score is calculated for each gene by calculating the discrete cross-correlation between the genetic wave and the target distribution. The ten members with the highest fitness are chosen to continue through to the next generation.

FIGURE 5.



Sample curves and correlation values. From top to bottom, correlation values: 0.5773, 0.4082, 1.0000

FIGURE 6.



A sample gene with two sinusoidal components. Enemy ship placement is the purple wave, with the gene's summation as the green wave: the two components are the red wave on the bottom and the smooth arch wave. Correlation is 0.702624

Reproduction occurs between every two genes in the selected set of genes. This process produces 45 (ten choose two) new genes, but is applied twice: once with mutation and once without. The selected set of ten genes is also carried through unchanged to the next generation, giving the next generation a total of 100 members.

Reproduction between two members is called crossover. In this problem, since the genes form a gestalt, the method of crossover must be based on the entire wave formed by all the harmonics of the gene [5], rather than a simple interchange of its harmonics. Crossover between the aggregate waves is a problem that requires some mathematically complex analysis. First the average of the two waves are computed, which is a perfect child, but is composed of ten sine waves. To reduce the component count to five, Fourier analysis is employed to isolate the five harmonics with the highest amplitudes, which are kept as the components of the child gene. This allows for the best approximation to the perfect child curve that is only composed of only five harmonics. To create the mutated population, a copy of each crossover gene is mutated, which consists of introducing random coefficients to some or all of the harmonics. Each coefficient has a 25% chance to be mutated. The Fourier analysis performed during crossover can be done by a Fast Fourier Transform (FFT) algorithm, which has a complexity of O(n logn); this reduces analysis to a reasonable run-time.

The problem space is extremely large, since each of the twenty coefficients of the gene are parameterized over a large range. However, this is not an important limitation, since crossover can be shown to converge on the solution in a reasonable number of generations [5]. To ensure that this genetic algorithm implementation does not converge on a local maximum, random coefficient mutations are introduced during the creating of each new generation.

This genetic algorithm runs a large number of generations at the start of each game after the first. Also, in each game after the initial game; one or more generations are run per turn. The member with the highest fitness in the most recent generation is used when the sinking logic (described in the following section) calls for a random location.

This shooting algorithm also incorporates a filter which will never allow the algorithm to return a square that cannot possibly contain a ship. This means that a checkerboard pattern is applied for targeting, which will adjust according to the size of the largest enemy ship remaining.

C. Sinking Logic

Sinking logic is the linear instruction logic needed when a ship has been hit, or "locked"; it governs following successive shots, attempting to sink the entire ship. A simple algorithmic solution exists, described below, but has a fundamental weakness. The opponent may place ships in certain patterns to make sinking more difficult, such as placing ships in parallel, but an intelligent system can resolve these issues.

A simple sinking algorithm exists that is based on a three state finite state machine. The states are "searching", "locked", and "sinking". Searching is the action of attempting to locate

an enemy ship position. When a shot is successful, hitting a ship, the state is changed into "locked". In this state, the original shot position is saved and successive shots are placed in the positions above, below, left, and right of that saved position. This is to determine the direction of the ship. Once a successful shot has been made in this state, the direction is saved and the state is changed to "sinking".

In this "sinking" state the following shots are placed one unit ahead, based on the saved direction, from the original shot position until a miss. This will shoot the length of the ship until going past the end of the ship. If the ship is not yet sunk the algorithm reverts back to the original shot that lead us into the "locked" state and continues shooting, one unit ahead from the last, in the opposite direction until the ship is sunk.

This hard-coded solution works well against random ship placement, but may be optimized to adapt to intelligently placed ships. The hard-coded solution may not perform well against special ship placement patterns, such as an intersecting "T" shape. If an enemy were to more likely place ships in certain directions, such as vertically, the hard-coded solution does not adapt and optimize for this. A quickly varying system with enough logic to implement the above algorithm, as well as new instructions to extend functionality, is needed. Genetic programming, an application of genetic algorithms onto an instruction set, has the ability to create many varying solutions containing solution logic. Linear genetic programming has been proven to be a valid approach for linear logic representation [10] in a genetic programming environment. Such a system has already been implemented [7] with the worst performing solution as efficient as the standard hard-coded algorithm. Since genetic programming has been helpful for representation of finite state machines [1], the solution structure is split into three sub states that are three different blocks of instructions. Each block must explicitly call a "jump" instruction to move into the next state.

A formal set of instructions are needed for solution representation. This set must be simple enough for crossover and mutation, but also complex enough to implement the basic algorithm. Extra instructions are added to give the potential of more dynamic, and perhaps surprising, solutions. For simplicity, the language is of a linear paradigm without the concept of looping, instruction redirection, or branching; thought it does implement conditional state change. Symbolic regression and standard genetic programming representations are not used due to their limitation in terms of efficient linear logic representation [10]. No concept of variables exists except for a register machine, a series of variables used by the solution at run-time. These variables have default values when the gene is initialized, but carry over during state changes. Each register and instruction is defined in Figure 7.

FIGURE 7.

Variable Name, Variable Description

- **TargetPos**, The current target location, defaults to (0, 0)
- **TempPos**, Temporary position, defaults to (0, 0)
- **TargetDir**, The current target direction, defaults to North / up
- **TargetHit**, Boolean where true if the last shot was successful in hitting a ship
- **TempHit**, Temporary boolean flag for internal usage

Instruction Name, Instruction Description

- **Target**, Find a position, placing into TargetPos, to shoot at; Managed by the targeting genetic algorithm in the above section
- **Shoot**, Shoot at the target position, from TargetPos. Sets TargetHit to true on a successful hit, sets TargetHit to false on failure or out of bounds
- **MoveFwd**, Move the target forward, from TargetPos, based on current direction, from TargetDir, and places the new target into TargetPos
- **RandDir**, Sets, into TargetDir, a random direction. Manages internally to prevent shooting at already-shot locations.
- **VertDir**, Change to face a random vertical direction, placing into TargetDir. Manages internally to prevent shooting at already shot locations.
- **HorzDir**, Change to face a random horizontal direction, placing into TargetDir. Manages internally to prevent shooting at already shot locations.
- **SavePos**, Save the current position from TargetPos into TempPos.
- **LoadPos**, Loads the current TempPos position into TargetPos.
- **SetTrue**, Set the current TempHit to true
- **SetFalse**, Set the current TempHit to false
- **IfHit**, Execute the next line if TargetHit is true, else, jump to the line after that
- **IfMiss**, Execute the next line if TargetHit is false, else, jump to the line after that
- **IfTrue**, Execute the next line if TempHit is true, else, jump to the line after that
- **IfFalse**, Execute the next line if TempHit is false, else, jump to the line after that
- **Nop**, No operation; Used as a "blank" instruction

For clarification a sample problem with the above instruction set is written in Figure 8. The left column includes the instructions with the right including line-by-line descriptions. Each row is representative of the three different states.

FIGURE 8.

Sample solution; Comments are written in C-style "//"

Targeting state (5/10)

1. Target
2. SavePos // Needed for the next state
3. Shoot
4. IfHit
5. Jump // Jump to the next state

Locking state (10/20)

1. LoadPos
2. VertDir // Check up down
3. MoveFwd

```
4.  IfHit
5.  Jump          // Jump to the next state
6.  LoadPos
7.  HorzDir       // Check left right
8.  MoveFwd
9.  IfHit
10. Jump          // Jump to the next state
```

Sinking state (9/20)

```
1. MoveFwd      // Keep walking through it
2. Shoot
3. IfMiss       // If we went past the end
4. IfTrue       // And we have already
                // sunk the other side..
5. Jump         // Go back to seeking
6. IfMiss       // If we went past the end (but have
                // not yet sunk the other side...)
7. OppDir       // Flip directions
8. IfMiss       // Same logic as above
9. LoadPos
```

The algorithm used for managing the genetic programming aspects is outlined in Figure 9 followed with a discussion of each step. The gene is represented as a block of 50 instructions. This block is subdivided into three sections of 10, 20, 20 representing the three possible states. These blocks are not continuous and are only accessible with the "jump" instruction. The first block is small since the targeting logic is usually very simple (see Figure 8). Instructions always remain at the top most possible index, with the remaining space filled with the "no-op" instruction. This is to prevent poor gene crossing or unnecessarily large code. Each section represents, in order, the Targeting, Locking, and Sinking states. Such a design, based on its crossover function, has enough of a convergence condition to result in more efficient solutions over time [5].

FIGURE 9.

1. Randomly create an initial population when starting a new game
2. For each round that is not the first round
    a. Repeat the following for a set 20 times
    b. Measure the fitness of each gene
    c. Select the top 10% of this list
    d. Randomly breed the top 10% and grow the population until the population pool is full
3. Return the best individual gene

If the opponent is a new player, the base population is constructed from a pre-trained population with slight mutations. This pre-trained population comes from a completely randomized population that is trained by hard-coded competitors beforehand. Games played by the same opponent keep their previous population. The maximum population size is 20 genes. Due to the computer industry's movement towards multi-core systems, a larger population might be more plausible as breeding would be split among many cores [2].

Once this base population is initialized or carried from a previous game, a selection is performed both to find competitive genes for reproduction as well as to find weak genes for termination. A fitness function is applied to each gene and a fitness value is assigned. All genes are then sorted and the top 10% are selected for reproduction. The remaining 90% are removed from the population. This selection method requires a function to measure the correctness of a gene.

A fitness function allows for a measurable way to find the efficiency, or correctness, of a given solution. Each gene needs to be tested for its success rate as well as for correct logic. The fitness function returns low values for the most successful genes, directly representing the number of shots needed to sink a ship. Genes with low success return high fitness values, as do logically invalid genes such as those with infinite looping. This value is generated by running each solution through all previously saved games from the current opponent. Shot placements, from the "target" instruction, are consistent between each fitness function call so that no variance occurs between genes in this measurement function.

Once a group of genes are selected a reproduction method is needed to generate new solutions to fill the population pool. Reproduction is complex in nature due to the need to intelligently merge possible solutions. This is further complicated because the gene data structure contains three states, or three sub-sections of code, that are independent.

To counter this problem, a chunk-swapping algorithm with slight mutation is used. This method is the same as a cross-over combination between both mates, with slight differences. Instead of swapping many single instructions at different locations, a single large randomly-sized chunk is swapped at a different location per mate. A random size from one to half the maximum sub-section size of the gene is selected. A random chunk of the same size is location from the other parent, then swapped. Both of these new children have a series of mutations applied. In this solution a low 10% probability of single instruction mutation per sub-section of code is reasonable since the chunk-swapping algorithm is a sufficient convergence condition [5].

Once the population pool is filled to its maximum size with new genes, another selection is applied in which only the top performing gene is returned as the best known solution for the upcoming game against the current competitor.

The maximum population size, as well as program size, are relatively small. This is needed as any larger sizes would grow the run-time significantly without a major improvement in creating efficient solutions. In such a case where a larger population is needed, work for fitness calculations, reproduction, and simulations can be reduced by splitting it amongst different cores. As the computer hardware industry shifts to multi-core processors, such designs can give great speed reduction benefits [2]. Another optimization could be to extend the instruction set to be more complete, though this would grow the solution complexity exponentially.

IV. Analysis

To empirically test the above solutions a simulation environment was created. Several hard-coded solutions were created as competitors against the above defined solution.

Each sub-component was tested individually as well as the entire solution was tested as a whole. Different board sizes as well as a range of games per round were chosen. This environment was programmed in C++ and is valid for both UNIX, Unix-Like systems, as well as WIN32. The analysis tools are available online at

http://code.google.com/p/battlestar-ai/

Several hard-coded solutions were created to test each individual component in this solution as well as attempt to observe possible opponent ship placement and targeting patterns. Hard-coded players included random placement of ships, random shooting, patterned shooting, and hard-coded sinking logic.

For each solution paired against itself, there was a significant increase in chances of winning for the first player, usually close to 10% for the first player. This shows that a large advantage is given to the player who shoots first. In an adaptive solution more hits also allow for patterns to be recognized in the opponent's ship placement, which is vital to long-term victory. The checkerboard pattern targeting was also significantly more efficient than random shot placement, with a difference of 70% to 30%. Any hard-coded players with sinking logic were extremely efficient with almost 90% chance of winning against solutions that do not implement sinking logic.

V. Conclusion

This paper has presented the outline of an artificially intelligent Battleship player that combines preloaded training data and data acquired in previous games to create a dynamic counter-strategy for any opponent. By utilizing intelligent placement methods and maintaining a record of the opponent's firing patterns, this solution places ships in areas of the board which the opponent has neglected in past games. The offensive targeting system, based on genetic algorithms, quickly adapts to take advantage of emerging patterns in the opponent's placement strategies. Once a hit is achieved, the player uses sinking logic which has been optimized against previous placement patterns by the same opponent for a swift demise of the wounded vessel.

References

[1] K. Benson, "Evolving Finite State Machines with Embedded Genetic Programming for Automatic Target Detection". Congress on Evolutionary Computation, 2000, vol. 2, pp. 1543-1549. Jul, 2000.

[2] S. M. Cheang; K. Sak; K. H. Lee, "Evolutionary parallel programming: design and implementation". Evolutionary Computation, vol. 14, n.2, pp. 129-156. Jun. 2006.

[3] F. Dellaert; D. Fox; W. Burgard; S. Thrun, "Monte Carlo Localization for Mobile Robots". IEEE International Conference on Robotics and Automation (ICRA99). May, 1999.

[4] IEEE Intelligent Systems staff. "Genetic Programming". IEEE Intelligent Systems, vol. 15 n. 3, pp.74-84, May 2000.

[5] Q. C. Meng; T. J. Feng; Z. Chen; C. J. Zhou; J. H. Bo, "Genetic Algorithms Encoding Study and A Sufficient Convergence Condition of GAs". Systems, Man, and Cybernetics, 1999. IEEE SMC '99, vol. 1, pp. 12-15, Oct. 1999.

[6] N. Metropolis; S. Ulam, "The Monte Carl Method". Journal of the American Statistical Association, vol. 44, pp. 335-341, Sep. 1949.

[7] D. Monniaux, "An Abstract Monte-Carlo Method for the Analysis of Probabilistic Programs". ACM SIGPLAN Notices, vol 36, pp. 93-101. Mar. 2001.

[8] M. Mysinger, "Genetic Design of an Artificial Intelligence to Play the Classic Game of Battleship". Genetic algorithms and genetic programming and Stanford, pp. 101-110. 1998.

[9] A. Roy, "Artificial Neural Networks - A Science in Trouble". 2000 ACM SIGKDD, vol. 1, n. 2, pp. 33-38, Jan. 2000.

[10] T. Weise; M. Zapf; K. Geihs, "Rule-based Genetic Programming". Bio-Inspired Models of Network, Information and Computer Systems, pp. 8-15, Dec. 2007.