Battleships - A Game Playing Agent

I. Problem Description

Battleships is a 2-player guessing game originally published by the Milton Bradley Company in 1943. Each player uses two square grids of size 10, one to place the own ships, one to record the shots fired at the opponent. A total of 10 ships of various sizes has to be placed by every player in such a way that no two ships occupy adjacent squares. After the initial placement phase, players alternate in shooting at enemy ships. The goal of the game is to sink all enemy ships before one's own ships are sunk. An example of a board is shown in Figure 1.

The methods of solving guessing games that include a search component can have widespread real-world applications. Many robot-robot or robot-human interactions can be represented as adversarial games with uncertainty. Gaining insight into the structure and solution processes with toy problems such as Battleships will help us apply these methods to other problems of a similar structure.

II. Related Work

[1] introduces the problem of playing the game Battleships as a decision problem and proves that it is NP-complete. This paper, however, provides no insight into how a game playing agent can be implemented. [2] uses an approach based on genetic algorithms that adapts to an opponent's playing style over multiple games. The drawback of this method is the fact that the program has to complete a learning phase before it can play efficiently. [3] presents an extensive software design document from the initial requirements specification to the implementation of the game. The artificial intelligence is rudimentary using simple random initial placements of the ships and random shots. [4] presents a Java-based implementation using a strictly rule-based agent. These rules are hard-coded nested if-else constructs. However, such hard-coded approaches tend to be inflexible and ill-suited to react to unforeseen conditions.

III. Implementation

We split the problem into two sections: the ship placement and the actual game playing. In our hypothesis, the best possible placement is one that maximizes entropy. As a result, our algorithm places the ships randomly on the board. Specifically, for each ship, the program finds and stores all of the valid positions (up to $2n^2$ on an $n \times n$ board) to place the ship on the board, given the locations of ships that have already been placed. It then chooses one of these valid positions at random. For randomness, we used the built-in C pseudo-random generator rand(), seeded with the system time at the beginning of the run of the program.

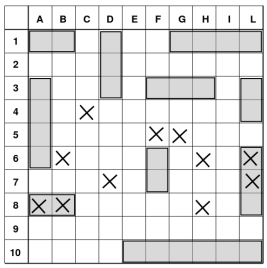

Fig. 1. Example of a Battleships board.[1]

Unlike the previous methods outlined in Section II, we implemented an approach that is more sophisticated than simply choosing random shots and that requires no knowledge of an opponent's playing style, or previous knowledge. Instead of choosing a move randomly, our algorithm directly attempts to determine which locations are most likely to contain a ship, given the *belief state* we are in as a result of the knowledge we have obtained from previous attacks.

Our approach is to use Monte Carlo sampling. This technique has been used in conjunction with game trees for several games with imperfect information, including Scrabble and Bridge [5], and more recently has been used in an AI framework known as Monte Carlo Tree Search to achieve success with AI for the classic board game Go, which proved resistant to other AI techniques, as well as with modern board games such as Settlers of Catan [6]. However, we felt that for Battleships, Monte Carlo sampling alone (without any game tree) would provide a strong AI player.

In detail, our algorithm samples the *belief state* of the locations of the opponent's ships N times. It then chooses to attack the currently empty location that contains a not-yet-hit part of a ship in the largest number of these samples.

Sampling the belief state is itself not a trivial task. We accomplished it as follows:

- Compute a list of the valid board locations for ships of each size.
- Attempt to place the ships on the board in order from largest to smallest.
- For the current ship:
  1) Choose a location randomly from among the valid board locations which cover the most possible hits currently on the board.
  2) If the location doesn't conflict with previously placed ships, push it onto the stack of placed

[1] http://en.wikipedia.org/wiki/File:Battleship_game_board.svg

ships. Otherwise, remove it from the list of valid locations.

3) If the list of valid locations becomes empty, pop the top placed ship off the stack, reset the valid board locations for all other unplaced ships, and begin trying to place the just-popped ship instead.

This algorithm is by no means a perfect sampling of the belief state. In fact, on certain occasions, it will fail to cover all of the known hits, thus producing a sample not actually in the belief state. However, this occurs very rarely, in less than 1 in 1000 samples in practice, although in certain very rare belief states it may occur with probability close to or even equal to 1. Nevertheless, in most belief states it is a good approximation. Moreover, truly perfect sampling would require computing the entire belief state in order to learn the probability of a ship covering any particular location. For an $n \times n$ board with s ships, the number of belief states will be at most $(2n^2)^s$. For the classic version of Battleships (10×10 board, 5 ships), this would be 320 billion states, which is rather infeasible.

IV. Evaluation

In this section we will outline the reference implementation presented in [4] that served as our main point of comparison. The source code is available for free and enabled us to integrate the AI approach into our implementation. We will describe the game strategy and artificial intelligence used in [4] and how we measured the performance of our Monte Carlo-based agent with respect to the rule-based artificial intelligence shown in [4] and against a completely randomly shooting agent.

We evaluated our implementation by competing against the reference implementation provided in [4]. Since [4] provided a Java-based implementation, our first approach was to interface our C++-based implementation with it through sockets, a common way of achieving language agnostic interprocess communication. As it turned out though, [4] lacked one key features that would make automated testing impossible. Ships had to be placed manually unlike in our implementation, which features automated, randomized placement of ships. Since the reference implementation used object-oriented programming and isolated their AI in its own class, we ported it to C++ and integrated it into our implementation. The main idea, a rule-based approach of playing Battleships using hard-coded conditional statements, was kept, although we optimized the provided code significantly.

In its essence, the AI in [4] uses a pattern database to search for ships in absence of any recent hits and a method that seeks to sink a ship once it is found. The pattern database causes the algorithm to shoot in diagonal lines until it either finds a ship or completes all shots stored in the pattern database. At this point, it continues shooting randomly to fill all left gaps. The patterns and the order of priority are shown in Fig. 2, where the green positions are tried before blue (top triangles filled) and red ones (bottom triangles filled). Once an enemy ship is found, the algorithm searches in the neighborhood, i.e. all adjacent squares until the ship is sunk. We will show in this section and Section V that our algorithm exhibits superior performance in finding ships and ending the game successfully.

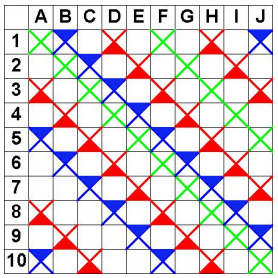

Fig. 2. Pattern database of reference implementation [4].

TABLE I
Results for N = 10 and 100 Trials

	Win Rate	Avg.	Min.	Max.
Monte Carlo	85 % / 88 %	90.17 / 87.96	74 / 75	98 / 96
Rule-based	15 %	88.16	86	92
Random	12 %	90.48	86	97

Table I and II summarize our quantitative results with regard to win rate and average, minimum, and maximum number of shots to win over a hundred trials. Note that the first row contains two datasets separated by slashes. The first refers to competing against the rule-based agent, whereas the second one refers to competing against the random agent. Based on these numerical results, the win rate does not appear to be correlated to N, the number of samples of the belief state in the Monte Carlo simulation (see Section III). One would expect to achieve a higher win rate as N increases. The contrary is the case, the win rate is 85 % for N = 10 and 75 % for N = 100. We assume that this is an artifact of a single competition containing 100 trials where the win rate has not converged to its true value. On the other hand, one can clearly see the effect of increasing N in the decrease of the average and minimum number of shots to win (from 90.17 to 84.92 and from 87.96 to 81.67 respectively).

Fig. 3 and Fig. 4 show the number of shots required to win as a function of trial i. In both figures, the red line marks the trials conducted with N = 10, the black lines refer to trials with N = 100. Solid lines indicate the measured values, dashed lines indicate the mean over all won trials. The data presented in both figures is the same as shown in Table I and Table II, where Fig. 3 summarizes data obtained by competing against a rule-based agent and Fig. 4 summarizes data obtained by competing against a random agent. Note that the number of won trials is not the same for both datasets in each figure. The shorter dataset has been padded with the value of the last won trial in each figure.

V. Discussion

Whereas search seems to be an appropriate tool for completely observable and deterministic environments, as

TABLE II
RESULTS FOR N = 100 AND 100 TRIALS

	Win Rate	Avg.	Min.	Max.
Monte Carlo	75 % / 81 %	84.92 / 81.67	57 / 63	100 / 99
Rule-based	25 %	87.8	83	99
Random	19 %	93.56	86	100

well as for certain adversarial games, it became clear that search was not the tool of choice for partially observable games such as Battleships. After our initial tries to model Battleships as $A*$ search with chance nodes and alpha-beta pruning and a heuristic that takes probabilities into account, we realized that the number of different belief states was simply too large. Moreover, unlike many games, we realized that Battleships is amenable to a greedy strategy. Specifically, an attack which results in a hit is *always* better than an attack that results in a miss, whereas in games such as chess a move which provides large immediate benefit can end up being a poor move in the long run. As a result, we felt the best strategy was simply to determine where on the board an attack would most likely be successful, which led us into the realm of probability and probabilistic reasoning. As it turned out, Monte Carlo sampling methods proved to be suitable tools for dealing with partially observable environments.

As mentioned in [7], the Monte Carlo approach is part of a family of algorithms commonly referred to as randomized sampling algorithms, specifically direct randomized sampling algorithms. Each variable is sampled in order of size of the ships and the probability distribution is conditioned on already assigned values, in our case known hits and misses. As with any sampling algorithm, we expect the frequency of specific events (e.g. finding a ship of size s at location (x, y)) to converge to the sampling probability. Therefore, we would expect to achieve better results as we increase the number of samples N. As shown in Table I and Table II we can see such an improvement in the minimum and average number of shots it takes to win a game (see Fig. 3 and Fig. 4 for details by trial). These numbers drop significantly as we increase N from $N = 10$ to $N = 100$. While the win rate decreases as N increases, this may be due solely to the small sample size.

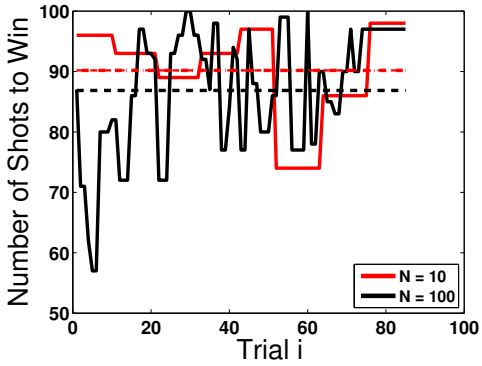

Fig. 3. Shots required to win against a rule-based agent.

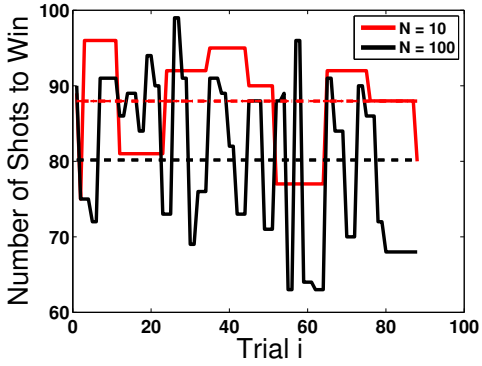

Fig. 4. Shots required to win against a randomly acting agent.

Finally, while we are pleased with our results, there are several potential improvements and investigations suggested by our work. First, other than setting a relative limit of each move taking no more than 15 seconds on our machine, we were not particularly concerned with efficiency. Our algorithm performed well within that limit on classic Battleships: with 100 Monte Carlo samples, the longest move we recorded was 200 ms, with most moves taking no more than 40 ms. However, it would be interesting to see how our algorithm performs regarding skill and efficiency in a more generalized version of Battleships, with, say, a 50×50 board and 100 ships. While our code can handle such an input, we have not yet tested it.

Second, as mentioned in Section III, there are some hit/miss layouts where our sampling algorithm will almost never manage to choose a valid sample; that is to say, it will nearly always place the ships in a configuration where they fail to cover every known hit on the board. This situation occurred very rarely in our tests. However, it is a weakness in the algorithm which could potentially be exploited by an adversary with clever initial ship layout. It would be interesting to see if we could find a way to exploit this potential weakness. Alternatively, it would be nice to find a way to modify our algorithm to remove this problem without sacrificing efficiency.

REFERENCES

[1] M. Sevenster, "Battleships as a decision problem," *ICGA Journal*, vol. 27, no. 3, pp. 142–149, 2004.

[2] J. G. Bridon, Z. A. Correll, C. R. Dubler, and Z. K. Gotsch, "An artificially intelligent battleship player utilizing adaptive firing and placement strategies," available Online. www.cores2.com/files/FinalResearchPaper.pdf.

[3] S. L. Andersen, "Battleship - design and implementation," 2007, thesis.

[4] A. Ahmed, R. Brown, M. Colmer, D. Harton, K. Doran, and S. Pickford, "Artifical intelligence battleship game," Available online at http://code.google.com/p/battleshipaiproject/, source Code only.

[5] I. Frank, D. A. Basin, and H. Matsubara, "Finding optimal strategies for imperfect information games," in *AAAI/IAAI*, J. Mostow and C. Rich, Eds. AAAI Press / The MIT Press, 1998, pp. 500–507.

[6] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-carlo tree search: A new framework for game ai," in *AIIDE*, C. Darken and M. Mateas, Eds. The AAAI Press, 2008.

[7] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Pearson Hall, 2010.