Optimal Strategies against a Random Opponent in Battleship

Maxime Audinot*[1], François Bonnet†[2], and Simon Viennot‡[2]

[1]Département Informatique et Télécommunications, ENS Rennes, France
[2]School of Information Science, JAIST, Japan

Abstract: Battleship is a two-player game, where each player tries to guess the positions of the opponent's ships. In this paper, we consider a simplified sub-problem, by assuming that the opponent places the ships randomly. Our goal is to compute the optimal deterministic strategy that sinks the ships with the smallest average number of shots. First, we describe algorithms to compute this exact minimal average number of shots. Our implementation on small grids allows us to show that greedy strategies are not always optimal. The usual grid used in the real game is too big for computing the exact optimal strategy, so in the last part of the paper, we show how to compute lower and upper bounds of the optimal average number of shots.

1 Introduction

Game of Battleship Battleship is a famous two-player game played worldwide, where each player tries to guess the positions of the opponent's ships. The game can be played with a pencil and paper. More sophisticated versions with plastic boards have been commercialized mainly in Europe and America, but also in Japan, under the name "re-da- sakusen game".

The rules are simple. Each player is given two square grids. One of the grid is used to place a set of ships and should be hidden from the opponent, and the other one is used to record the successful or missed shots targeted at the opponent. The ships are placed by the players on their grid at the beginning of the game, like for example on Figure 1, and cannot be moved thereafter. The usual grid size is 10×10, with a set of 5 ships of lengths 2, 3, 3, 4, and 5.

Alternately, the players shoot at their opponent's grid, trying to find on which cells the ships are placed. The opponent announces after each shot if it was a "hit" or a "miss", i.e. if it was successful or not. When all the cells of a ship have been shot, the ship is said to be "sunk", a fact that is announced or not depending on the version of the game. The winner is the first player to sink all the ships of his opponent.

In this paper, we consider the main variant of Battleship where no information is announced apart from "hit" and "miss". Moreover, adjacent ships, even in diagonals, are forbidden.

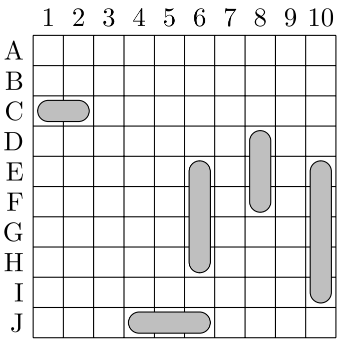

Figure 1: Possible start position chosen by one player

Related work Despite the international popularity of Battleship, to the best of our knowledge, there exists only a limited amount of research. The game has possibly suffered from its childish image, which is unfortunate because it contains many interesting and difficult sub-problems. In 1988, Rodin published a strategy for sinking the opponent's ships, and tested it with human players [4]. In 2003 and 2008, Sakuta and Iida investigated randomized strategies both for the placement of the ships and for the sinking attacks [5], [6]. They also used the occurence matrix, a concept that we detail in section 2.3. In a research report of 2009, Bridon et al. applied genetic algorithms to adapt to the ship placement of the opponent [2]. Battleship can be described as a Partially Observable

*maxime.audinot@ens-rennes.fr
†f-bonnet@jaist.ac.jp
‡sviennot@jaist.ac.jp

Markov Decision Process (POMDP), and as such, it was used in 2010 by Silver and Veness [8] as a testbed for a new class of Monte-Carlo algorithms applied to POMDPs that have attracted much attention.

Game against a random opponent A game of Battleship can be decomposed into two sub-games played in parallel with no interaction. In each sub-game, the opponent places the ships, and the main player tries to sink them as quickly as possible. In this paper, we consider only this sub-game of Battleship, and as in [8], we also suppose that the ships are placed randomly and uniformly on the grid by the opponent.

Contrary to previous works where the strategies are mainly evaluated by simulations, we are interested in the theoretical optimal deterministic strategy, that allows to sink the opponent's ships as quickly as possible. Our goal is to compute the theoretical exact value of the average number of shots required by an optimal strategy. This direction of study is the equivalent for Battleship of numerous previous works on the game of Mastermind, for example the famous result of Koyama and Lai [3].

Organization of the paper First, we describe in section 2 an algorithm to compute the number of ship configurations and the frequency matrix. In section 3, we define the concept of deterministic strategy and we detail how to compute the optimal strategy that sinks the ships in the smallest average number of shots. We show that the optimal strategy is sometimes strictly better than any greedy strategy. Finally, in section 4, we compute lower and upper bounds of the optimal strategy for the usual 10×10 game.

2 Configurations and Counting

2.1 Configuration

We start our study of Battleship by counting the number of different configurations of ships. A configuration of ships is simply defined as any distribution of the ships on the grid, like on Figure 1. However, two different distributions of ships on the grid can occupy exactly the same set of cells, and in that case care is needed to define what we consider as *different* configurations.

In this paper, adjacency of ships is forbidden, so the only way for two distributions of ships to occupy the same set of cells is when the positions of the two ships of size 3 are exchanged. There is no reason to distinguish the two ships of size 3, so even if we exchange the position of the ship D8-E8-F8 with the ship J4-J5-J6 on Figure 1, we consider that it is the same configuration.

2.2 Configurations count

Algorithm 1 shows how to count the number of possible ship configurations on a given grid. We suppose that the ships are indexed from 1 to the number of ships, and the algorithm places each ship on the grid recursively. The parameters in the recursive calls are the index of the next ship that should be placed on the grid, and the current state of the grid with already placed ships. The first recursive call is done with an initial index of 1 (place the first ship), and an empty grid. Algorithm 1 and our C++ implementation are directly inspired by a program from Scherphuis [7].

```
Output: number of ship configurations
Function CountConfig()
    return Count(1, empty grid);

Input: index $i$ of the next ship to place
Input: $pos$, the current board position
Function Count($i$, $pos$)
    $nbConfigs \longleftarrow 0$;
    foreach position on the grid of the ship $i$ do
        if position of ship $i$ is valid then
            if all ships are placed then
                $nbConfigs \longleftarrow nbConfigs + 1$;
            else
                $newPos \longleftarrow pos +$ ship $i$;
                $r \longleftarrow$ Count($i+1$, $newPos$);
                $nbConfigs \longleftarrow nbConfigs + r$;
            end
        end
    end
    if $i$ = 1 and two ships of size 3 are used then
        return $nbConfigs/2$;   // final result
    else
        return $nbConfigs$;
    end
```

Algorithm 1: Count the ship configurations

The test of *position of ship i is valid* consists in checking that the new ship i that we place on the grid is not intersecting with or adjacent to any already placed ships. For the sake of implementation simplicity, the two ships of size 3 are distinguished in the main part of the algorithm. They have different indexes, so each configuration will be counted twice.

229713268	290312632	358949313	382685666	395191893	395191893	382685666	358949313	290312632	229713268
290312632	276237938	307909560	304268668	302249855	302249855	304268668	307909560	276237938	290312632
358949313	307909560	339346322	334820797	333795527	333795527	334820797	339346322	307909560	358949313
382685666	304268668	334820797	329017778	329162179	329162179	329017778	334820797	304268668	382685666
395191893	302249855	333795527	329162179	331435930	331435930	329162179	333795527	302249855	395191893
395191893	302249855	333795527	329162179	331435930	331435930	329162179	333795527	302249855	395191893
382685666	304268668	334820797	329017778	329162179	329162179	329017778	334820797	304268668	382685666
358949313	307909560	339346322	334820797	333795527	333795527	334820797	339346322	307909560	358949313
290312632	276237938	307909560	304268668	302249855	302249855	304268668	307909560	276237938	290312632
229713268	290312632	358949313	382685666	395191893	395191893	382685666	358949313	290312632	229713268

Figure 2: Occurrence matrix of the 10×10 grid

We simply divide by 2 the final result (when the function returns from the first call). The result of Algorithm 1 on the usual 10×10 grid is 1,925,751,392 different configurations of ships, a value already known, for example in [1].

Algorithm 1 is easily generalized to count the number of possible configurations that are compatible with a set of hit and miss shots. Compatibility with previous miss shots is included in the test of validity for a ship position, and compatibility with previous hit shots is checked just after all ships are placed.

2.3 Frequency matrix

A useful information for analyzing the game is the frequency matrix, which gives the frequency of occupation of each cell of the grid, assuming that the opponent chooses a configuration of ships randomly and uniformly. It is straightforward to adapt Algorithm 1 to compute the occurrence matrix, which gives for each cell of the grid, the number of configurations where a ship occupies the given cell. The frequency matrix is then obtained by dividing the occurrence matrix by the total number of configurations.

11.9	15.1	18.6	19.9	20.5	20.5	19.9	18.6	15.1	11.9
15.1	14.3	16.0	15.8	15.7	15.7	15.8	16.0	14.3	15.1
18.6	16.0	17.6	17.4	17.3	17.3	17.4	17.6	16.0	18.6
19.9	15.8	17.4	17.1	17.1	17.1	17.1	17.4	15.8	19.9
20.5	15.7	17.3	17.1	17.2	17.2	17.1	17.3	15.7	20.5
20.5	15.7	17.3	17.1	17.2	17.2	17.1	17.3	15.7	20.5
19.9	15.8	17.4	17.1	17.1	17.1	17.1	17.4	15.8	19.9
18.6	16.0	17.6	17.4	17.3	17.3	17.4	17.6	16.0	18.6
15.1	14.3	16.0	15.8	15.7	15.7	15.8	16.0	14.3	15.1
11.9	15.1	18.6	19.9	20.5	20.5	19.9	18.6	15.1	11.9

Figure 3: 10×10 frequency matrix in percentages

Figure 2 gives the occurrence matrix for the usual game on the 10×10 grid, and Figure 3 gives the corresponding frequency matrix, in percentages. For example, it means that a corner of the grid is occupied by a ship in 11.9% of the configurations (229,713,268, precisely).

In [5], Sakuta and Iida computed the occurence matrix when adjacency is authorized, and showed that cells close to the center are occupied more frequently. Compared to this previous result, the result of Figure 2 and 3 when adjacency is forbidden is unexpected. The highest occupation frequency is not found in the center, but in the middle of the edges, with 20.5%. Even more surprisingly, the frequency decreases to 15.7% when going one line in direction to the center, before increasing again to 17.3%.

3 Exact optimal strategies

3.1 Deterministic strategies

In this paper, we consider only deterministic strategies to sink the opponent's ships. If we play many times against the same opponent, a deterministic strategy should be avoided because it would be easily countered by the opponent. However, in order to simplify the analysis, we consider here that the opponent cannot learn our strategy and cannot adapt the placement of his ships, so we will not discuss the problem of randomized strategies, and we restrict the analysis to deterministic strategies.

A deterministic strategy consists in choosing a first cell to shoot at, then choosing a second cell depending on the result of the first shot, hit or miss, and so on. A deterministic strategy is completely described as all the choices of cells done after each hit or miss. The natural way to represent such deterministic process is a decision tree, where each node represents the choice of the cell for the next shot, and each edge represents the result of the shot, i.e. a hit or a miss.

By convention, we represent a hit by an edge to the left, and a miss by an edge to the right. On the figures, we have also drawn small circle marks on edges to the left and small cross marks on edges to the right to make it clear that they correspond to hits and misses.

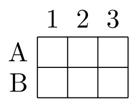

Figure 4: 2×3 grid (with a ship of size 2 to find)

For example, let consider the grid of Figure 4. We suppose that the opponent has placed one single ship of size 2 on this grid, and we want to sink it. Then, Figure 5 shows the decision tree of a possible deterministic strategy to sink that ship. The first shot is done at A2. If it is a hit, the second shot is done at A1. If it is a miss, the second shot is done at B2, and so on.

After two hits, for example at A2 then at A1, the ship of size 2 is sunk. We indicate this situation by a checkmark. There is no need to shoot anymore, so this a leaf of the strategy tree. Each leaf of the tree corresponds to one possible position of the ship of size 2 on the grid. From left to right on Figure 5, the leaves correspond to the ship placed on A1-A2, A2-B2, A2-A3, B1-B2, B2-B3, A1-B1, A3-B3.

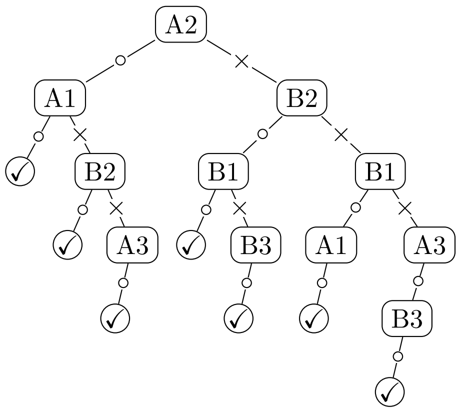

Figure 5: Decision tree of a deterministic strategy

The decision tree of Figure 5 has an height of 5, and an average height of $(2+3+4+3+4+4+5)/7 = 25/7$.

For our study, it is convenient to also define the notion of *total height* of the tree, as the sum of the heights of all branches from the root node to a leaf. For the tree of Figure 5, the total height is $2+3+4+3+4+4+5 = 25$.

3.2 Optimal strategy computation

In this paper, our goal is to find the deterministic strategy with the smallest average number of shots to sink the ships, which is equivalent to minizing the average height of the associated decision tree. We denote this minimal average height by $\phi(n, m, L)$ for a grid of size $n \times m$ and a list L of ship lengths, or simply $\phi(n, m)$ when L is the usual list of five ships.

Since the number of leaves of the strategy tree is fixed (equal to the number of configurations), minimizing the average height of the tree is equivalent to minimizing the *total height* defined in the previous section. We describe below how it is possible to minimize the total height of the tree with a recursive procedure. The advantage of minimizing the total height is that it can be done by manipulating only integer numbers.

From a grid G where some hit and miss shots have already been made, we start by considering all the possible choices of cells c for the next shot. Then, for each choice of cell c, we compute the total height $H(G, c)$ of the optimal tree where c is the root node, and finally we keep as the final answer the cell c that gives the smallest total height. The recursion step appears in the computation of $H(G, c)$, which can be broken in three steps :

- compute the total height $H_{hit}(G, c)$ of the optimal left sub-tree after a hit shot on c
- compute the total height $H_{miss}(G, c)$ of the optimal right sub-tree after a miss shot on c
- deduce the total height $H(G, c) =$ $H_{hit}(G, c) + H_{miss}(G, c) + countConfig(G)$

The formula in the last step comes from the fact that when we unite the left sub-tree and the right sub-tree to obtain the final tree, the length of each branch of the left sub-tree and each branch of the right sub-tree is increased by 1. The total effect is similar to adding the number of leaves of the tree, hence the number of ship configurations.

Size / Ships	4×4	4×5
2	$146/24 \approx 6.1$	$218/31 \approx 7.0$
2,3	$878/104 \approx 8.4$	$2326/242 \approx 9.6$
3,3	$196/24 \approx 8.2$	$595/67 \approx 8.9$
2,3,4	-	$994/88 \approx 11.3$

Table 1: Average number of shots of the optimal strategy

Algorithm 2 gives the complete description of this recursive procedure. Unfortunately, this algorithm is extremely slow, because it roughly considers all the possible orderings of cells for the shots. To accelerate the algorithm, we use transposition tables, and avoid shooting on meaningless cells where no ship can appear, but even with these improvements, only small grids can be computed in a reasonable time. Table 1 shows the exact ϕ values that we could compute for some grid sizes and set of ships .

3.3 Greedy Strategy

A natural heuristic to sink the ships as quickly as possible is simply to shoot at the cell with the highest probability of finding a ship. We call *greedy strategy* any strategy that consists in shooting successively on cells with the highest probability of finding a ship. Such a cell can be found by computing the frequency matrix described in 2.3, and choosing the cell with the highest frequency value.

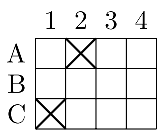

Figure 6: Counterexample to the greedy strategy

It must be noted that usually there are multiple cells with the same highest frequency value, so that there is no unique greedy strategy. Anyway, it is natural to expect that at least one of the greedy strategies is optimal. If it was true, it would allow us to reduce greatly the number of candidate cells in Algorithm 2, and compute the optimal strategy for much bigger grids. It is true in many situations, but unfortunately - and surprisingly - not always. By using Algorithm 2, we were able to find counterexamples where the optimal strategy is strictly better than any greedy strategy.

Figure 6 shows such a counterexample. This is a 3×4 grid with already two missed shots, and the goal is to find a single ship of length 2. Figure 7 shows the tree of the best greedy strategy (which can be obtained by hand on this small grid), and Figure 8 the tree of the optimal strategy, computed with Algorithm 2.

The greedy strategy shoots first at B3, for an average height of 54/12, whereas it is in fact strictly better and optimal to shoot at B2, for an average height of 53/12. The difference is tiny, which confirms that a greedy strategy is an excellent heuristic, but the neg-

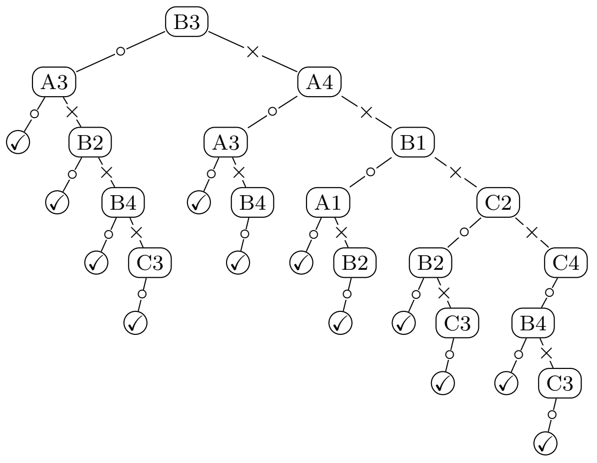

Figure 7: Best greedy strategy for the counterexample

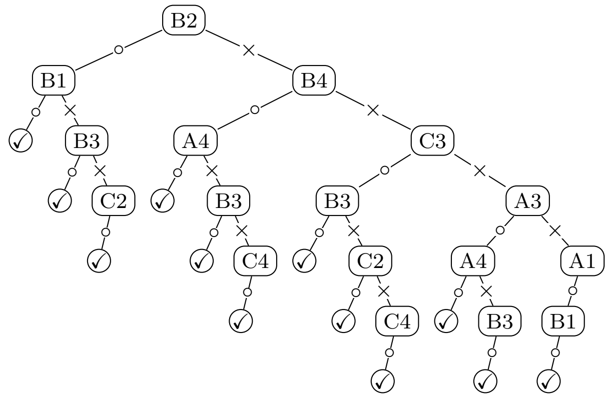

Figure 8: An optimal strategy for the counterexample

ative result is that greedy strategies cannot help us to compute the optimal strategy.

4 Bounds for the 10×10 grid

The exact algorithm is too slow to compute $\phi(10, 10)$ on the usual grid of size 10×10 with 5 ships. Instead, we try to compute lower and upper bounds.

4.1 Lower bound

A lower bound for the optimal strategy can be derived by analyzing the structure of the associated decision tree. First, we know that the decision tree is a binary tree. Since the result of a shot can only be a hit or a miss, each node is at most of degree 2. Secondly, we can count the number of leaves of the decision tree. Each possible leaf of the tree corresponds to one single

```
Input: hitShots, list of positions of hit shots
Input: missShots, list of positions of miss shots
Output: Sum of heights of all branches from the root to a leaf
Function ComputeOptimal(hitShots, missShots)
    $nbConfigs \longleftarrow$ CountConfig(hitShots, missShots);
    if $nbConfigs = 0$ then
        return 0;
    end
    $currentMinHeight \longleftarrow Max$;
    foreach empty cell c of the grid do
        $afterHitShots \longleftarrow$ Merge(hitShots, c);
        $afterHitHeight \longleftarrow$ ComputeOptimal(afterHitShots, missShots);
        $afterMissShots \longleftarrow$ Merge(missShots, c);
        $afterMissHeight \longleftarrow$ ComputeOptimal(hitShots, afterMissShots);
        $currentHeight \longleftarrow nbConfigs + afterHitHeight + afterMissHeight$;
        if $currentHeight < currentMinHeight$ then
            $currentMinHeight \longleftarrow currentHeight$;
        end
    end
    return $currentMinHeight$;
```

Algorithm 2: Compute the optimal average number of shots

set of cells where a ship was hit.

When adjacency of the ships is forbidden by the rules and when the two ships of size 3 are not distinguished, a set of hit cells corresponds exactly to one configuration of ships. For example, in the case of the usual 10×10 grid, it implies that all deterministic strategies, including the optimal one, are described by a decision tree with 1,925,751,392 leaves. If adjacency is authorized, it is more difficult to count the number of leaves because different configurations of ships can lead to the same leaf in the decision tree.

It is well-known that for a given number of leaves, the binary tree of smallest height is obtained for a complete binary tree, not necessarily full. In our case, we are not interested by the tree of smallest height, but by the tree of smallest average height. Fortunately, this does not make any important difference, and it is straightforward to prove that the binary tree of smallest average height is again a complete binary tree.

Figure 9 shows an example of complete binary tree for the case of 11 leaves. The average height of this tree is $(4 * 6 + 3 * 5)/11 = 3 + 6/11$, which is the smallest possible average height of any binary tree with 11 leaves.

In the case of the battleship game on the usual 10×10 grid, we have $\ell(10\times10) =$ 1,925,751,392 leaves. Since the number of nodes of a complete and full tree of height 30 and 31 is repectively $2^{30} = 1,073,741,824$ and $2^{31} = 2,147,483,648$, we know that the height of any binary tree containing $\ell(10 \times 10)$ leaves is 31, with all leaves of height at least 30. This gives us a first lower bound of $\phi(10, 10) > 30$. In the next paragraph, we improve this lower bound by counting precisely the number of leaves of height 30 and height 31 in the complete binary tree.

In general, if we want to construct a complete binary tree with ℓ leaves and $2^k < \ell < 2^{k+1}$, there are $\ell - 2^k$ leaves that need to be added to the complete and full tree of height k. For each leaf that is added, we need to expand one leaf of height k into two leaves of height $k + 1$. Thus, it will result in $2 \times (\ell - 2^k)$ leaves of height $k+1$ and $2^k - (\ell - 2^k)$ leaves of height k. The average height $\overline{h}$ of this complete binary tree is given by equation 1, which simplifies in equation 2.

$$\overline{h} = \frac{2\ell - 2^{k+1}}{\ell} \times (k+1) + \frac{2^{k+1} - \ell}{\ell} \times k \qquad (1)$$

$$\overline{h} = k + \frac{2\ell - 2^{k+1}}{\ell} \qquad (2)$$

We check the correctness of equation 2 on Figure 9. We have $\ell = 11$, $k = 3$, so equation 2 gives an average height of $3 + (2 * 11 - 16)/11 = 3 + 6/11$, which is indeed the result that we obtained by hand above.

In the case of the battleship game on a 10×10 grid, we have $\ell = 1,925,751,392$, $k = 30$, so equation 2 gives $\overline{h} \approx 30.8$ and we can conclude that $\phi(10, 10) > 30.8$.

Table 2 compares the lower bound of equation 2 to the exact ϕ value, for the small boards where the exact value is computable. If we compare the case of two ships of size 3 on boards of size 4×4 and 4×5, we see that the lower bound is relatively better on the board of size 4×5, i.e. the board with a lot of empty cells. One reason is that if the ships occupy a large proportion of the board, there are many cases for which we shoot the ships while knowing perfectly their position on the board. In such cases, the nodes in the decision tree are of degree 1 instead of 2, which leads to an average height notably higher than a complete binary tree.

Ships	Size	lower bound	exact ϕ
2	4×4	≈ 4.6	≈ 6.1
	4×5	≈ 4.9	≈ 7.0
2,3	4×4	≈ 6.7	≈ 8.4
	4×5	≈ 7.9	≈ 9.6
3,3	4×4	≈ 4.6	≈ 8.2
	4×5	≈ 6.0	≈ 8.9
2,3,4	4×5	≈ 6.5	≈ 11.3

Table 2: Comparison of lower bounds with known exact ϕ values

4.2 Upper bound

Finally, we compute an upper bound of the optimal average number of shots needed on the usual 10×10 grid. Since the average number of shots achieved by any given strategy is an upper bound of the average number of shots needed by the optimal strategy, the straightforward way to obtain an upper bound consists in choosing any strategy, computing the number of shots needed by this strategy on each of the 1,925,751,392 possible configurations of ships, and taking the average. The number of 1,925,751,392 configurations is not so high, and our implementation takes only around 15 minutes to compute the upper bound associated to a given strategy, on a 3 Ghz computer.

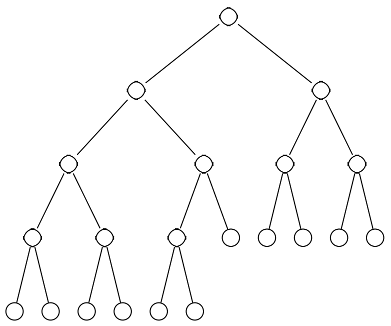

Figure 9: A complete binary tree with 11 leaves

The main problem in this approach is to define and represent a strategy. It is not convenient here to define a strategy as in section 3.1 with a complete deterministic tree, because of the high number of nodes. Instead, we define a strategy as an algorithm, which acts as short representation of the associated deterministic tree.

A first and naive strategy consists in shooting the cells one after the other, from left to right, and from the top line to the bottom line, until all ships are sunk. Such a strategy is obviously not very efficient, and no human player would play like that, but it is easy to implement. It gives us a first upper bound of 91.7 shots.

49		9		1		5		43	
	47		35		39		31		44
12		13		19		15		32	
	38		29		25		16		6
4		22		23		26		40	
	42		28		24		20		2
8		18		27		30		36	
	34		17		21		14		10
46		33		41		37		48	
	45		7		3		11		50

Figure 10: Order of the shots for the upper bound

We describe now a much better strategy that will give us a smaller upper bound. Since the smallest ship is of size 2, we can ensure that all ships will be found by shooting to only half of the cells of the grid. An efficient order of the cells consist in using the frequency matrix of Figure 3, which gives roughly the order of cells of Figure 10.

The complete strategy consists in shooting to the cells in the order of Figure 10, and whenever a ship is found, destroying it completely by shooting along the vertical and horizontal lines starting from the hit cell. The shots on each of the four vertical and horizontal directions are done until a miss is obtained. The only case where we stop earlier the clearing of the four directions is for the last ship. In that case, the strategy stops as soon as the total number of hits needed to sink all the ships (17 hits) is obtained. We computed that this strategy takes on average 58.7 shots.

An improvement is possible by clearing first the horizontal directions, and then clearing the vertical directions only if no hit was achieved on the horizontal directions. We obtained an average of slightly less than 55.6 shots, which is our current best upper bound: $\phi(10, 10) < 55.6$.

Figure 11 shows the distribution of the number of configurations that require a given number of shots to sink the ships. The distribution on the right corresponds to the naive strategy and the one on the left to the more elaborate strategy of our current best upper bound. This strategy of the best upper bound has a standard deviation of 7.6 shots and a worst case of 73 shots.

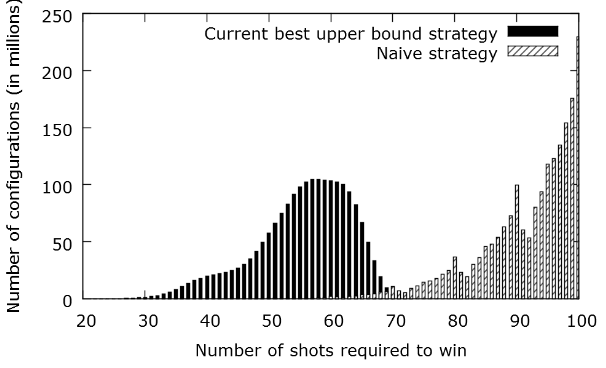

Figure 11: Distribution of the number of configurations in function of the number of required shots

5 Conclusion and Future Work

In this paper, we have described algorithms to compute the optimal deterministic strategy against a random player in the game of Battleship. We obtained the exact optimal average number of shots needed on some small grids. The most difficult exact result that we obtained is an average number of shots of 994/88 for the 4×5 grid with three ships of size 2, 3 and 4.

Then, we have computed a lower bound of $\phi(10, 10)$, the optimal average number of shots for the usual game on a grid of size 10×10, with a theoretical analysis of the deterministic strategy trees. Finally, we have also obtained an upper bound for the usual game, by defining a strategy and computing its corresponding number of shots on all of the 1,925,751,392 possible configurations of the ships. It allows us to conclude that $30.8 < \phi(10, 10) < 55.6$.

Computing the exact value of $\phi(10, 10)$ does not seem feasible in a near future, but it is an interesting challenge to try and reduce further the gap between the lower and upper bounds. Especially, the upper bound could be improved relatively easily by considering more detailed and complicate strategies.

References

[1] Battleship game combinatorics. `http://www.haskell.org/haskellwiki/Battleship_game_combinatorics`, 2012.

[2] J. Bridon, Z. Corell, C. Dubler, and Z. Gotsch. An artificially intelligent battleship player utilizing adaptive firing and placement strategies. Technical report, The Pennsylvania State University, 2009.

[3] K. Koyama and T. Lai. An optimal mastermind strategy. *Journal of Recreational Mathematics*, 25:251–256, 1993.

[4] E. Y. Rodin, J. Cowley, K. Huck, S. Payne, and D. Politte. Developing a strategy for "battleship". *Mathematical and Computer Modelling*, 10:145–153, January 1988.

[5] M. Sakuta and H. Iida. Evaluation of attacking and placing strategies in the battleship game without considering opponent models. In *Proceedings of 1st International Forum on Information and Computer Technology*, pages 80–85, 2003.

[6] M. Sakuta and H. Iida. Decision making based on the generation of possible positions in an imperfect-information game - a case study using the battleship game. *Journal of Game Amusement Society*, 2:28–35, 2008. (in Japanese with an English abstract).

[7] J. Scherphuis. C# source code for counting battleship configurations. `http://forums.xkcd.com/viewtopic.php?f=17&t=101584&view=previous#p3186886`, 2012.

[8] D. Silver and J. Veness. Monte-carlo planning in large pomdps. In *Proceedings of the 24th Annual Conference on Neural Information Processing Systems*, pages 2164–2172, 2010.