Optimal Strategies on Attack and Defense in the Game of Battleship

Chanbin Park

Handong International School

ABSTRACT

Battleship is a well-known strategic guessing game played by two players. The players first place their ships and then continuously try to sink every ship of the opponent. In this paper, the act of placing ships is defined as "defense", and the act of guessing ships is defined as "attack". The goal of this paper was to find good attacks against bad defense and good defense against good attack strategies. First, two good attack strategies, hunt and target/probability density strategy, are introduced and are compared with a randomly guessing strategy. Especially, when implementing the probability density strategy, using bit optimization resulted in a significantly faster algorithm than a naive implementation. Then, in the final section, several good defense strategies against the two good attack strategies are shown.

1. INTRODUCTION

Game of Battleship. Battleship is a strategic guessing game for two players. It was first played as a pencil and paper game dating in World War I. Each player has two boards composed of 10 × 10 grids; one to position the player's ships and to check the opponent's guess, and the other to record the player's guess. There are 5 ships, which are Carrier, Battleship, Cruiser, Submarine, Destroyer, and they have sizes of 5, 4, 3, 3, and 2 grids, respectively. Ships need to be positioned only horizontally or vertically and cannot share a grid. Unlike other board games such as chess and checkers, Battleship is not a combinatorial game because it is played with imperfect information (i.e. the initial ship configuration is hidden to the opponent).

Before the game starts, both players secretly position their ships on their first board. Then, each player takes turns to guess a target square on the opponent's board. Each of them has to tell the other whether the guess "hit", "missed", or "sank" a ship. If a ship sank, the name and size of the ship should be revealed. Whoever first sinks all of the opponent's ship wins the game[1]. However, there are a lot of variants to this game. For example, the rule of a well-known variant is only revealing if the opponent has hit or missed a ship, unrevealing if a ship has sunk.

Design of the Research. For simplicity, I designed the Battleship game so that only the information of "hit" and "miss" are allowed to the players. I am going to define guessing the ships as "attack", and placing the ships as "defense." So, according to this definition, the Battleship is a two-player game setting a defense strategy in the beginning and continuously attacking each other. Thus, there are two factors in the game, which are the attack/defense of each player. An attack can be evaluated as good, or bad. This applies to defense, as well. Thus, I defined the four factors shown below.

- Good Attack: Guessing ships according to the opponent's defense
- Good Defense: Placing ships according to the opponent's attack
- Bad Attack: Guessing ships randomly
- Bad Defense: Placing ships randomly

Design of the Paper. Section two is about good attacks when played against bad defense. I made the computer to generate a game board with randomly placed ships to implement bad defense. Section three is about good defense against good attacks. I created several ship configurations which I assumed to be good defense and experimented with good attacks discussed in section two. Finally, in the .conclusion part, I put the results altogether

2. "GOOD" ATTACKS AGAINST "BAD" DEFENSE

The basic outline for the calculation of the average number of turns an attack strategy took to finish the game is shown in algorithm. `numOfGames` represents the number of games that

1 Battleship Official Hasbro Rules, Rulebook insert for *Battleship* (2002 version)

the computer simulates, `numOfTurns` represents how many turns the attack strategy spent finishing each game, and the `totalTurns` is the sum of every `numOfTurns`.

```
function Example( )
    numOfGames ← 0, numOfTurns ← 0, totalTurns ← 0
    for each games do
        while true do
            make a guess and increment numOfTurns    ▷ making a guess differs by methods
            if every ship sank then
                totalTurns ← totalTurns + numOfTurns
                numOfTurns ← 0
                break
            end if
        end while
    end for
    avgNum ← numOfTurns/numOfGames
end function
```

Algorithm 1: Calculating average number of turns

2.1. Random Strategy

Implementation. Before I discuss some good attacks, I would like to introduce what a bad attack (random strategy) would look like. This algorithm randomly guesses grids until it hits every ship on the board. Note that random(a, b) returns a random value between a and b, both inclusive.

```
function GuessRandom( )
    numOfTurns ← numOfTurns + 1
    row ← random(0, HEIGHT - 1), col ← random(0, WIDTH - 1)
    while row, col is guessed before do
        row ← random(0, HEIGHT - 1), col ← random(0, WIDTH - 1)
    end while
    if a ship is hit then
        mark hit
    else
        mark miss
    end if
end function
```

Algorithm 2: Perform a random guess

2.2. Hunt and Target Strategy

Explanation. Basically, Hunt and Target strategy is a greedy solution that is used by many people. It has two stages of action. During the "hunt" stage, the computer guesses randomly until it hits a ship. Then, during the "target" stage, the computer guesses the squares adjacent to the recent shot in every direction until there is no more square that contains a ship. Thus this algorithm repeats the hunt and target stage until it sinks every ship. The row number starts from the top as 0, and the col number starts from the left as 0.

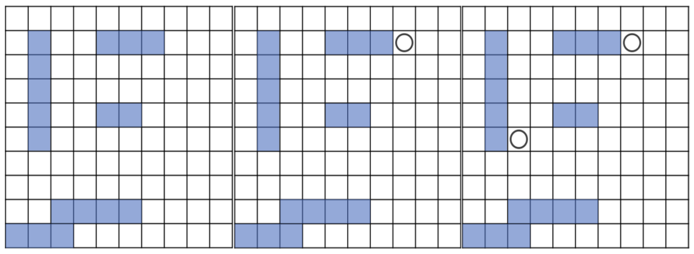


Figure 1: Hunt Stage

To visualize, as shown in Figure 1, when the computer is in the hunt stage, it guesses any grid randomly. The first picture shows the initial configuration of the board. The next picture shows that the algorithm guessed (1, 7), which is an empty grid. Then, it guesses an empty grid on (5, 2), remaining in the "hunt" stage.

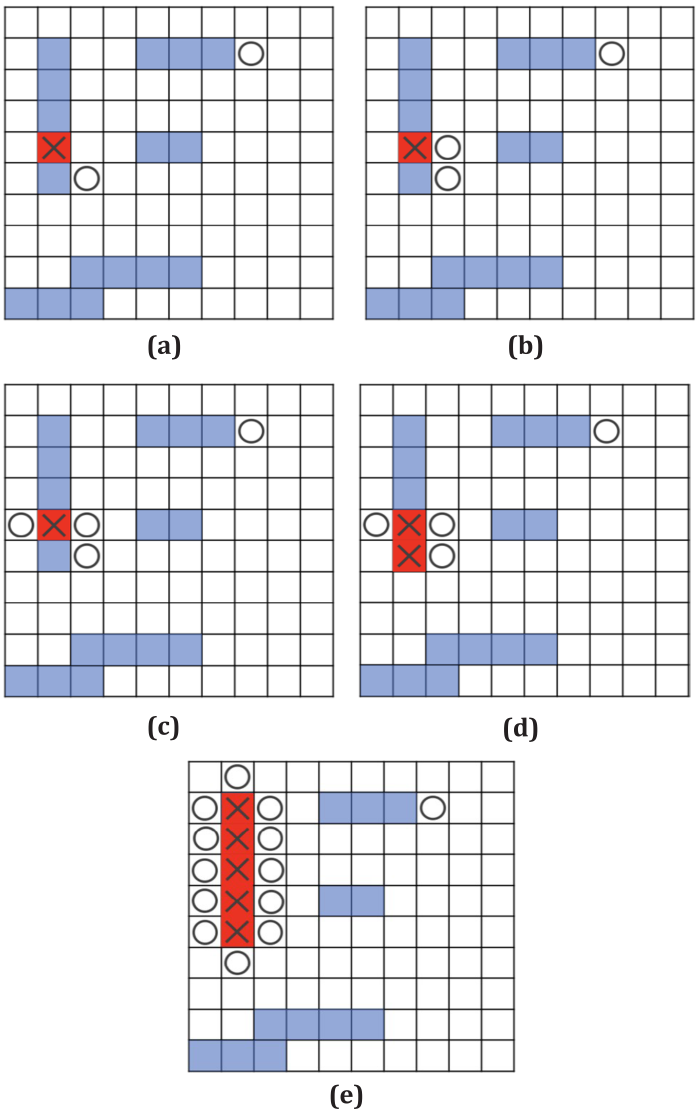


Figure 2: Target Stage

However, when a target is hit, shown in (a), the computer turns to the target stage. It searches for all the grids adjacent(not including grids connected diagonally) to the grid that is hit. As shown in picture (b), the computer has guessed (4, 2), an adjacent grid to the hit shown in (a). Then, it guesses another adjacent grid (4, 0) to the original hit, shown in (c). When the computer succeeds in hitting another part of the ship, like in (d), then it guesses all grids adjacent to that grid as well. When there are no other adjacent grids to guess, it returns to the hunt stage. Picture (e) describes the board after the algorithm returned to its hunt stage.

Implementation. Algorithm 3 shows the pseudocode for hunt stage.

```
function HuntAndTarget( )
    row ← random(0, HEIGHT - 1), col ← random(0, WIDTH - 1)
    while row, col is guessed before do
        row ← random(0, HEIGHT - 1), col ← random(0, WIDTH - 1)
    end while
    if a ship is hit then
        mark hit
        numOfTurns ← numOfTurns + 1
        SearchAdj(row, col)  ▷ a function that searches adjacent grids(algorithm 4)
    else
        mark miss
        numOfTurns ← numOfTurns + 1
    end if
end function
```

Algorithm 3: Perform hunt stage

In order to search for adjacent grids, I used depth-first search, shown in Algorithm 4.

```
function SearchAdj(row, col)
    for grids adjacent to (row, col) do
        numOfTurns ← numOfTurns + 1
        if grid has been never guessed then
            if the grid contains a part of a ship then
                mark hit
                if every ship sank then
                    return
                else
                    SearchAdj(nextrow, nextcol)
                end if
                if every ship sank then
                    return
                end if
            else
                numOfTurns ← numOfTurns + 1
                mark miss
            end if
        end if
    end for
end function
```

Algorithm 4: Perform target stage

2.3. Probability Density Strategy

Occurrence Matrix. Probability density strategy works based on an "occurrence matrix." An occurrence matrix is a matrix with the same size as the game board with numbers of every possible ship configurations. Each element of the matrix represents a grid with the number of an occurrence of a ship. Of course, the occurrence matrix will change after every turn or guess. For example, if a guess hits a ship, then the grid that was guessed must be included in every possible ship configuration for the succeeding calculations. If a guess misses, then the grid must be eliminated for every possible ship configuration.

2407483864	3458438788	4317794344	4774557212	5017010922	5017010922	4774557212	4317794344	3458438788	2407483864
3458438788	4292323972	4980720314	5348335096	5543394682	5543394682	5348335096	4980720314	4292323972	3458438788
4317794344	4980720314	5539435704	5841068044	6001718736	6001718736	5841068044	5539435704	4980720314	4317794344
4774557212	5348335096	5841068044	6121508012	6272257080	6272257080	6121508012	5841068044	5348335096	4774557212
5017010922	5543394682	6001718736	6272257080	6428054040	6428054040	6272257080	6001718736	5543394682	5017010922
5017010922	5543394682	6001718736	6272257080	6428054040	6428054040	6272257080	6001718736	5543394682	5017010922
4774557212	5348335096	5841068044	6121508012	6272257080	6272257080	6121508012	5841068044	5348335096	4774557212
4317794344	4980720314	5539435704	5841068044	6001718736	6001718736	5841068044	5539435704	4980720314	4317794344
3458438788	4292323972	4980720314	5348335096	5543394682	5543394682	5348335096	4980720314	4292323972	3458438788
2407483864	3458438788	4317794344	4774557212	5017010922	5017010922	4774557212	4317794344	3458438788	2407483864

Figure 3: Initial Occurrence Matrix

Figure 3 is the 10 × 10 occurrence matrix when the game starts (i.e. the information of hit and miss is void). There are 30,093,975,536 total configurations of placing 5 ships that have the size of 2, 3, 3, 4, 5 in a 10 × 10 board. As the color gets closer to red the number is bigger, and as the color gets closer to green, the number is smaller.

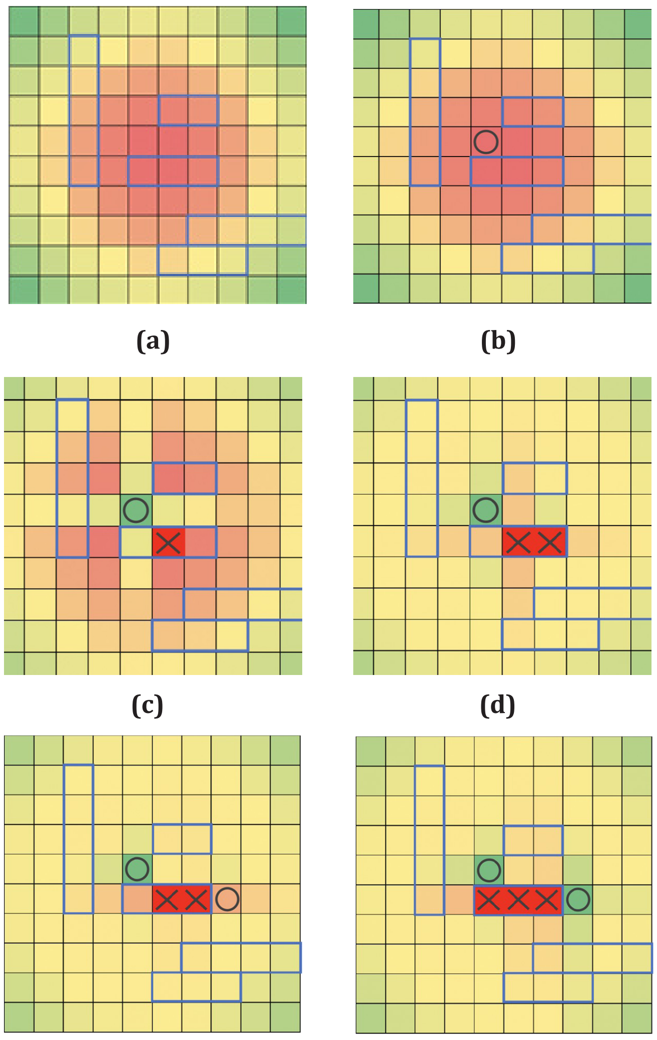


Figure 4: Probability density strategy

Explanation. Once the algorithm calculates the occurrence matrix, the algorithm will pick the grid with the highest number of occurrences and will update the algorithm based on if it was a hit or a miss. It would repeatedly build an occurrence matrix and guess until every ship sinks. Picture (a) in Figure 4 represents the initial configuration as well as the occurrence matrix. As shown in (b), the algorithm picks the grid with the biggest number in the occurrence matrix, which happens to be a miss. Then, the algorithm calculates the occurrence matrix based on this information and guesses the grid with the highest number again (shown in (c)). Then, it applies the information that the guess was a hit and calculates the matrix again. This process is repeated as shown in (d), (e), and (f). When a shot is missed, the grid turns to green, which represents 0.

Naive Implementation. The real challenge in implementing the probability density strategy is to implement the occurrence matrix. Therefore, in this section and the next section, the focus is on implementing the occurrence matrix. Here are some initial variables that we need for the naive implementation.

- `MainBoard[10][10]` // 2D array that keeps track of ships
- `ConditionBoard[10][10]` // 2D array that shows 1 if hit, 2 if missed, 0 if never guessed
- `OccurrenceBoard[10][10]` // 2D array that shows the number of occurence of each grid in every possible configurations(i.e. occurrence matrix)

In order to implement the occurrence matrix, I made a recursion function `Solver(int cur)` that can call itself recursively until every possible configuration is calculated. Note that `cur` is the index of the ship(0 $\rightarrow$ size 2, 1 $\rightarrow$ size 3, 2 $\rightarrow$ size 3, 3 $\rightarrow$ size 4, 4 $\rightarrow$ size 5) and `tempBoard[10][10]` is used to save the information of `MainBoard` temporarily.

```
function Solver(Cur)
    if Cur equals 5 then
        check the MainBoard with ConditionBoard
        if the information is consistent then
            reflect this information on the OccurrenceBoard
        else
            return
        end if
    end if
    tempBoard      Mainboard
    for every possible configuration of Cur^th ship do
        if it doesn't overlap with previous ship configurations then
            update MainBoard
            Solver(cur + 1)
            MainBoard ← tempBoard
        end if
    end for
end function
```

Algorithm 5: Perform Solver that calculates the OccurrenceBoard in a naive way

Optimized Implementation. The naive implementation takes a lot of time because it repeats things that can be calculated before or after the recursion function. Since there are 30,093,975,536 maximum ship configurations, time is crucial to this algorithm.

In order to optimize the probability density strategy, the first thing I did is to use a different coordinate system that can locate a ship. This allowed me to use advanced techniques such as bit masking. For example, for the naive implementation, I expressed the location of the ship with the left uppermost coordinate of a ship and the direction. For example, the three ships in Figure 5 would be (0, 0) with direction 1, (0, 1) with direction 1, and (0, 0) with direction 2. However, for the optimized implementation, I simply used a number starting from the left upper corner with horizontal configurations. When the horizontal configurations were all numbered, I moved on to the vertical ones. Therefore, in this case, the coordinates would be 0, 1, and 12. Of course, the total number of configurations(total locations) as well as the conversion to an actual coordinate system differ by every ship with different size.

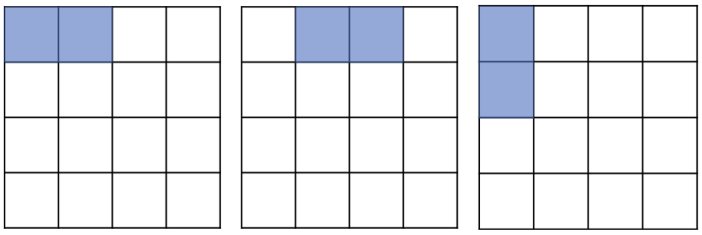


Figure 5: Coordinate System

If H = height of the board, W = width of the board, and S = size of a ship, the total number of possible configurations would be:

$$\text{totalConfigurations} = W * (H - S + 1) + H * (W - S + 1)$$

Having locations of ships as simple numbers, I put them into an array of Bitset with size 5 (total number of ships).

1	1	0	0	0	0	1	0	0	0	0	0	
0	1	2	3	4	5	6	7	8	9	10	11	...

Figure 6: Bitset

If the Bitset in Figure 6 belongs to the ship with size 2, 1 in the Bitset means that there can be a size 2 ship placed on location number 0, 1, and 6. Likewise, it means that there cannot be a size 2 ship on the location with a 0.

The main problems that take a lot of time for the naive implementation and how I managed to optimize it is shown in table 1. It is further clarified in algorithm 6.

	Naive Implementation	Optimized Implementation
Coordinate System	Used `Array[10][10]` to save the actual position of every ship	Used `Bitset[5]` to save whether there can be a ship in a location for every ship → `State[5]`
Calculating overlapping information	Had to place a ship on the board to see if it was occupied by a previous ship	Used `Bitset[5][TotalLocation][5]*` to calculate all the possible locations that ship "B" can be placed with ship "A" being placed on position "i" and up-dated the Bitsets of the succeeding ships → Possible[5][totalConfigurations][5] *Possible[ship "A"][position "i"][ship "B"]
Iterating through possible locations	Had to iterate through every possible configurations even if it could not be placed because there is a ship placed previously	Used destructive bitscan* to not look at the index of locations that overlap with previous ships *Destructive bitscan is a function which skips all the 0s in the bitset and returns the index of 1 and changes this index to 0. Returns -1 when there is no 1s left.
Comparing with ConditionBoard	Had to check if the 5 ship configuration matches with the information in ConditionBoard after positioning all 5 ships	(1) Saved the information of hits and misses from the ConditionBoard to `State[5]` prior to the calculation of the matrix (2) Calculated the number of hits that any index of location contains and saved it prior to the calculation of the matrix → `NumHit[5][totalConfigurations]` (3) Took the remaining number of hits as an argument of the `Solver` function and only referred to the prior calculation made in (2) to calculate the remaining number of hits
Adding information to OccurenceBoard	Had to add every information of the MainBoard to the OccurenceBoard if the MainBoard's information was consis- tent with the information of the ConditionBoard	Utilized the property of a recursive function: Returned 1 every time the final configuration was made and then added the total numbers of configurations to an external Array[5][TotalLocation]*. → `NumCount[5][totalConfigurations]` Added the numbers in the external array to the OccurrenceBoard after running the recursive function *Let's say that we want to know the number of configurations that include the ship size 2 in location 0. Then, we just need to refer to NumCount[index of ship size 2][0].

Table 1: Problems of Naive Implementation and Solutions

Thus, here are the initial variables and how I changed the recursive function.

- `State[5]` // Array of Bitset that keeps track of every ship configurations
- `Possible[5][totalConfigurations][5]` // Array of Bitset that saves information if a ship can be placed with another ship placed in a certain location
- `NumHit[5][totalConfigurations]` // Array of Bitset that saves information of number of Hits in the location of each ship
- `NumCount[5][totalConfigurations]` // Total Number of configurations in the location in each ship

```
function Solver(Cur, RemainHit)
    if Cur equals 5 then
        if RemainHit equals 0 then
            return 1        ▷ single successful 5 ship configuration made
        else
            return 0
        end if
    end if
    tempState ← State
    ret ← 0
    while true do
        i ← State[Cur].bitscan destructive()    ▷ performs destructive bitscan
        if i equals to -1 then
            break
        end if
        for every ship succeeding size Cur ship (nextShip) do
            State[nextShip]&= Possible[Cur][i][nextShip]
        end for
        x ← Solver(Cur + 1, RemainHit - NumHit[Cur][i]) ret ← ret + x
        NumCount[Cur][i] ←NumCount[Cur][i] + x State ← tempState
    end while
    return ret
end function
```

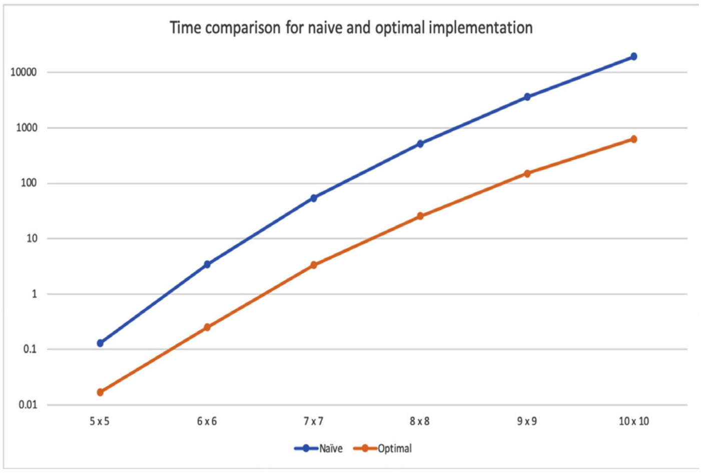
Algorithm 6: Perform Solver in a optimal way

Comparison. To evaluate the performance of the optimal implementation, I compared the time for both naive and optimal implementations that took to generate the initial occurrence matrix with 5 ships. Note that the gap between each implementation will further increase if we measure the time it takes to finish an entire game (i.e. calculating many occurrence matrices). As shown in table 2, for a 5 × 5 board, the optimized solution was about 7 times faster than the naive solution, and this value increases as the size of the board increases. It turned out that the optimized implementation calculated the initial occurrence matrix approximately 31 times faster than the naive implementation for a 10 x 10 board.

	5 × 5	6 × 6	7 × 7	8 × 8	9 × 9	10 × 10
Naive	0.13	3.4	54.4	519.9	3632.9	19418.7
Optimal	0.017	0.25	3.3	25.3	149.7	627.2

(units: seconds)

Table 2: Time Comparison between naive and optimal implementation



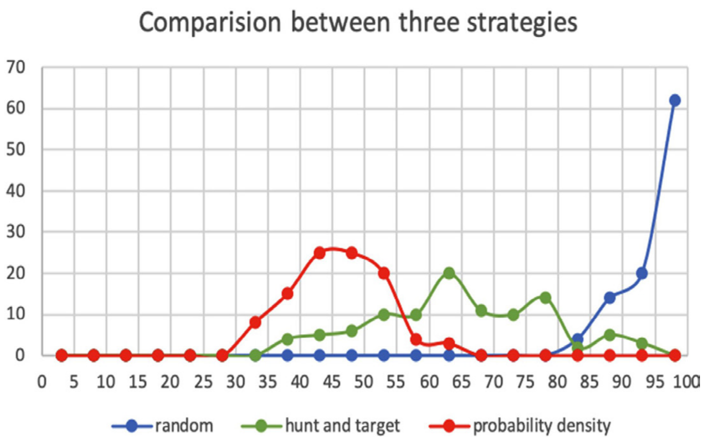
Graph 1: Time Comparison between naive and optimal implementation

2.4. RESULTS

I ran 100 times for all three strategies and the results are shown in table 3 as well as in graph 2.

	Average Turns	Minimum Turns	Maximum Turns
Random	95.4	81	100
Hunt and Target	65.9	39	94
Probability Density	46.1	32	65

Table 3: Comparison between three strategies



*The *x*-axis of the graph is the range of the number of turns. The point on each graph represents that there were games that took a number of turns in the range of the corresponding values on the *x*-axis. For example, the point that is between 5 and 10 turns has a value of 0, which means that there were 0 number of games that took between 5 and 10 turns to finish.

Graph 2: Comparison between three strategies

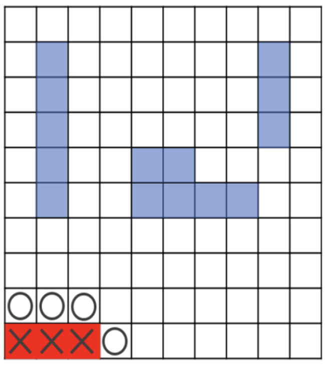
The random strategy, which I defined as bad attack, took an average of 95.4 turns to finish a game. Moreover, it took a minimum of 81 turns and a maximum of 100 turns. The hunt and target strategy's performance was substantially good compared to the random strategy. It took an average of 65.9 turns with a minimum of 39 and maximum of 94 turns. Lastly, the probability density strategy outweighed the hunt and target strategy. It took only an average of 46.1 turns, a minimum of 32, and a maximum of 65 turns. This data is reflected on graph 2. While the scatter plot for probability density graph is focused on the left side, the plot for the random strategy is focused on the right side. The plots of hunt and target strategy are concentrated in the middle.

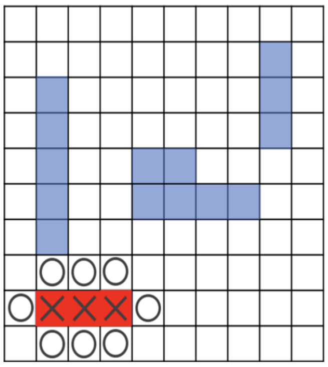
3. "GOOD" DEFENSE AGAINST "GOOD" ATTACK

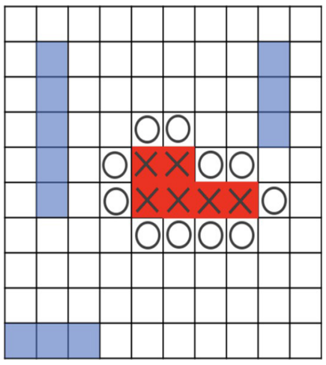
In this section, I created several models that assumed to be "good" defense and played it on "good" attacks determined in the previous section which are hunt and target, probability density strategy.

3.1. DEFENSE AGAINST HUNT AND TARGET STRATEGY

An interesting fact about the Hunt and Target strategy is that if two or more ships are adjacent to each other, the hunt and target strategy can identify all of these adjacent ships just by initiating target stage a single time. Moreover, if a ship is located on the edges of the board, the ship would be identified more quickly because the computer cannot guess grids that are outside of the board. For example, picture (a) in Figure 7 shows that a ship on the bottom left was identified in only 7 guesses because the algorithm does not look at grids that are outside the board. If the left bottom ship was not located on the edge, then the program would take 11 guesses to identify the same ship, shown in (b). Moreover, it only takes 17 guesses to identify the middle two ships in picture (c) when the two ships are next to each other. However, if they are separated, it takes a minimum of 22 guesses to identify both ships.


(a)


(b)


(c)

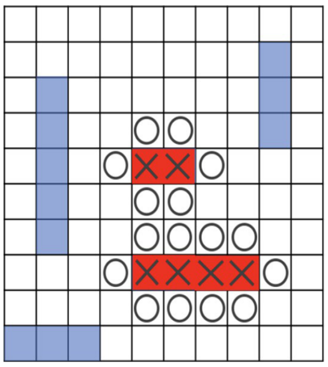

(d)

Figure 7: Ships located on edges and ships that are connected

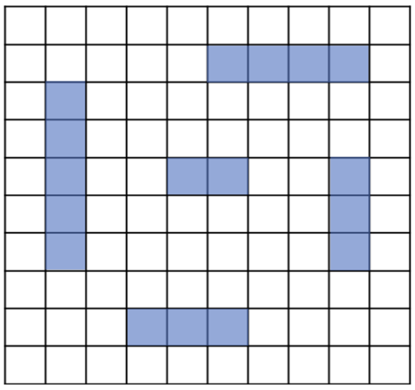
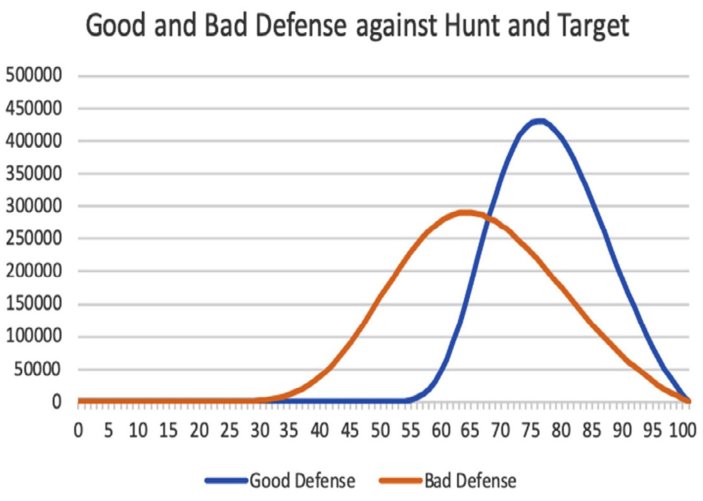


Figure 8: Good Defense on Hunt and Target

Considering these two factors, I created a model shown in figure 8 that has all ship separated as well as not located on the edges to make the hunt and target strategy slower. After 10,000,000 iterations, I got the following result, as expected.

	Average	Minimum	Maximum
Bad Defense	66.1	20	100
Good Defense	77.8	51	100

Table 4: Good and Bad Defense against Hunt and Target Strategy



Graph 3: Good and Bad Defense against Hunt and Target Strategy

While it took an average of 66.1 turns for the hunt and target strategy to finish a game against bad defense, it took approximately 12 more turns against my good defense. Moreover, while the minimum number of turns were 20 turns against bad defense, it turned out that the minimum was 51 for good defense, which is more significant because I iterated for 10,000,000 times. Overall, my defense strategy made the hunt and target strategy to struggle to sink all ships compared to the bad defense randomly generated by a computer.

3.2. PROBABILITY DENSITY STRATEGY

To make a good defense strategy, I experimented with 6 different types of initial configurations. There are two big factors for placing a ship, which are whether the ships are placed adjacent to each other or not, and whether the ships are placed on the middle of the board, placed on the edges, or placed evenly. Therefore, I created six models and experimented with them.

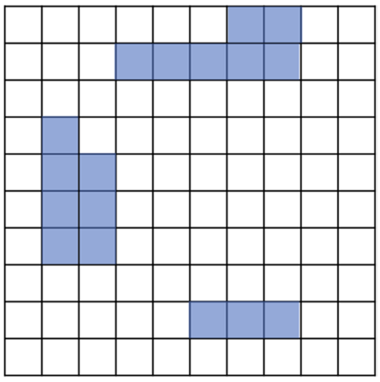
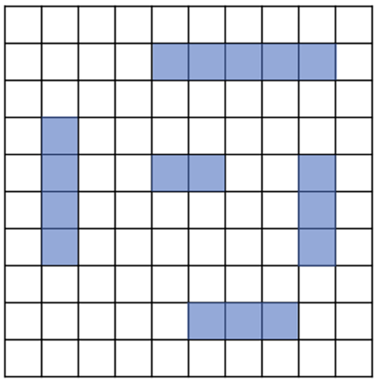
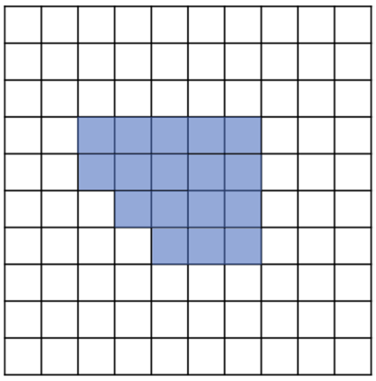
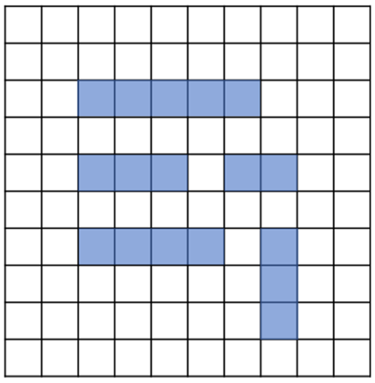
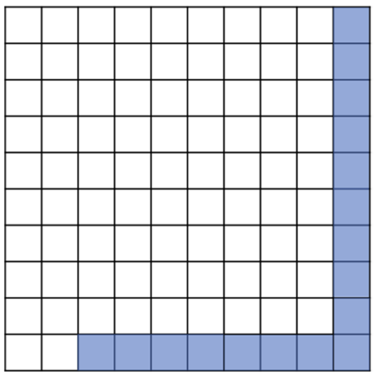
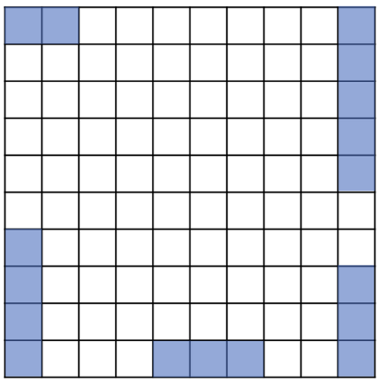
	Placed adjacent to each other	Not placed adjacent to each other
Placed evenly		
	53 turns	37 turns
Placed in the middle		
	26 turns	51 turns
Placed on the edges		
	35 turns	60 turns

Table 5: Six models for finding out Good defense against Probability Density strategy

Considering that the average number of turns the probability density strategy took against random placement of ships was 46.1, I found that a good defense strategy is to place all ships on the edges of the board without being adjacent to each other, which took 60 turns. Note that unlike hunt and target strategy, placing ships not adjacent to each other evenly on the board is not a good strategy against probability density (only took 37 turns).

4. CONCLUSION

Through implementing several attack strategies, it turned out that hunt and target strategy and probability density strategy were better strategies than the random strategy when playing against bad defense(randomly placing) taking 30 and 50 fewer average guesses, respectively.

When implementing the probability density strategy, I precomputed the information required for the calculation as well as used bit optimization in order to minimize the computing time. As a result, the algorithm was nearly 31 times faster in only calculating the initial occurrence matrix compared to a naive implementation.

Finally, I proposed some good defense against the hunt and target and probability density strategy and proved that it was 17.7%, 30.2% efficient, respectively.

Finding a mathematical optimal good defense strategy against a probability density attack strategy would be an interesting topic to study in the future. Furthermore, since the game of battleship is a two player game, it would be a great challenge to apply game theory to further explore optimal strategies on both attack and defense.

REFERENCES

1. Justin Castillo Anthony Cardenas. Battleship 254, 2015. URL https://github.com/natertater23/ BattleShip254.
2. Battleship. Datagenetics. URL http://datagenetics.com/blog/december32011/index.html.
3. C. Liam Brown. Battleship probability calculator: Methodology. URL https://cliambrown.com/battleship/methodology.php.
4. Jonathan Boyd Jacob Boyd. Battleship playing program utilizing probability density functions, 2016.
5. Audinot Maxime, Bonnet Francois, and Viennot Simon. Optimal strategies against a random opponent in battleship. *The 19th Game Programming Workshop, pages*, 2014:67–74, 2014.