CSE 202: Project Algorithmic Solutions

Martakis, Alex
amartakis@ucsd.edu A59012834

Sharan, Mayank
msharan@ucsd.edu A59012192

Choudhary, Twinkle
twchoudhary@ucsd.edu A59016512

Gupta, Satvik
sag005@ucsd.edu A59012252

March 15, 2023

1 Introduction

Our project is to model the game of battleship and solve algorithmic problems associated with it. In this document, we will review the modeling of the game and the algorithmic formulation of the problems we intend to solve. Further, we present algorithmic solutions to these problems and their mathematical analysis culminating in a unified game play strategy.

Battleship is played between 2 players. The objective of the game is to sink all ships of your opponent before they are able to sink yours. The gameplay has the following phases:

1. **Setup:** Both players arrange the ships they have on their grid.
2. **Guessing:** Both players take turns guessing a grid position of the opponent each turn
   - In this phase, if a guess is where an opponent ship is present, it is a ”hit”.
   - otherwise, it is a ”miss”.
   - A ship is sunk if all grids it is located in, are hit.
   - The player who sinks all ships of the opponent first wins.

Some key components of the game to be modeled are:

1. The board
2. The ships: count and length
3. The arrangement of ships
4. Player guesses and their outcome (”hit” or ”miss”)

In Section 2, we define the basic variables required to model the game which will be used in multiple algorithmic problems. We also discuss the restrictions we impose to limit the complexity of the problems we tackle. Sections 3 and 4 review the algorithmic problems and provide different algorithmic solutions with analysis

2 Game Modeling

2.1 Board Size

A typical game of battleship is played on a square board of size 10x10. However, to define more general problems and algorithms we will consider rectangular boards of arbitrary size $m \times n$, where m is the number of rows and n is the number of columns.

Without loss of generality we will assume $n \geq m$. This holds as any board with more row than columns can be rotated by 90° to get a board with more columns than rows. Thus, all configurations and actions have a rotational mapping.

Each grid position on the board will be indexed using a tuple of the form (x, y), where x is the row number and y is the column number. Further, $1 \leq x \leq m$ and $1 \leq y \leq n$.

(1, 1)					(1, 6)			
							(2, 8)	
	(3, 2)							
				(4, 5)				
								(5, 9)
		(6, 3)						
						(7, 7)		

Figure 1: A rectangular board of size 7 x 9 with grid positions illustrated

2.2 Ships

A standard battleship game has five ships of length 2, 3, 3, 4, and 5. In the spirit of generalization, we will take the number of ships and their sizes as parameters. We will denote the number of ships as n_s. We will restrict each ship to be 1 grid wide, so the only dimension they have is length.

$1 \leq n_s \leq m$, as we want there to be at least one ship. The upper limit to ensure that if needed, each ship can be put on separate rows or columns. The number of ships also contributes to the nature of the game. A sparse setup will likely have a very different approach than a dense one.

The length of the ships would be provided as an array *shipLen* of length n_s, indexed from 1 to n_s. $1 \leq shipLen[i] \leq m$ and $shipLen[i] \in \mathbb{Z}^+ \; \forall 1 \leq i \leq n_s$. This is to ensure that every ship can fit within any row or column it is placed on.

2.3 Arrangement of Ships

For any given ship with identifier i its location on the grid for player p is defined as a pair of tuples $(x^p_{is}, y^p_{is}), (x^p_{ie}, y^p_{ie})$, both valid coordinates on the board as defined in Section 2.1. For simplicity, we order the tuples so that $x^p_{is} \leq x^p_{ie}$ and $y^p_{is} \leq y^p_{ie}$. Here, $x^p_{is} = x^p_{ie}$ for a placement along a row or $y^p_{is} = y^p_{ie}$ for a placement along a column. We do not allow diagonal placement of ships as per the rules of the game. In case of a row placement $y^p_{ie} - y^p_{is} + 1 = shipLen[i]$ and in case of a column placement $x^p_{ie} - x^p_{is} + 1 = shipLen[i]$, and the other coordinates are equal. These tuples for each ship are stored in lists of tuples $shipS_p$ containing all (x^p_{is}, y^p_{is}) tuples and $shipE_p$ containing all (x^p_{ie}, y^p_{ie}) tuples. Both arrays are indexed by the ship identifier i.

Ships are not allowed to overlap so there does **NOT** exists any tuple (x, y) such that for any 2 ships i and j the following conditions are all true:

$$x^p_{is} \leq x \leq x^p_{ie}$$
$$x^p_{js} \leq x \leq x^p_{je}$$
$$y^p_{is} \leq y \leq y^p_{ie}$$
$$y^p_{js} \leq y \leq y^p_{je}$$

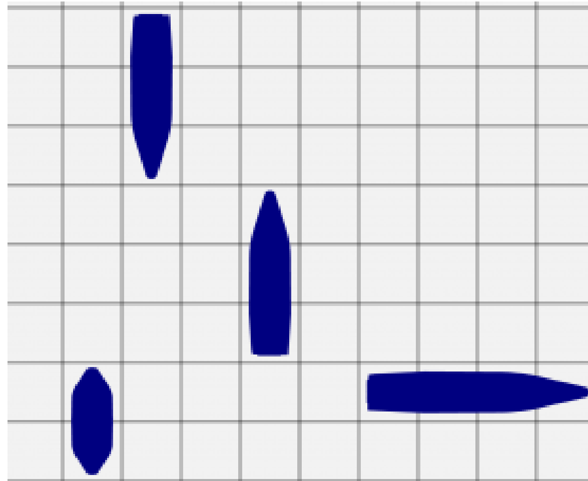

Figure 2: A sample arrangement of ships

2.4 Guess outcomes

We maintain 2 two-dimensional matrices to represent the guess status. For players p and q, these matrices are $Guess_p$ and $Guess_q$. Both matrices are of size $m \times n$ in correspondence to the board and are indexed starting from 1 so that the position tuples can be used as indices. $Guess_p[(x, y)]$ will provide the status on player q's grid of the guess by p at location (x, y) and vice versa.

Both matrices are initialized with all "U"s to indicate unexplored. As the game progresses in phase 2 the outcomes of all guesses by player p are updated in $Guess_p$ and by player q are updated in $Guess_q$. The value is updated to "H" in the event of a hit and to "M" in the event of a miss.

Both players have access to both matrices at all times as the information in them is available to both of them as the game progresses. We will not be storing the order of the guesses as it introduces storage complexity without adding any significant information and is not a reasonably expected for human players to be able to track.

3 Finding the First Hit

3.1 Problem Statement

We want to design a strategy that given a state of the game provides a sequence of guesses resulting in a hit. This can be used at the start of the game to locate the first ship or once a ship is sunk to find the next one. We define the problem with respect to the strategy of one player. This can be used by both players given the state of their corresponding $Guess$ matrix.

We find a sequence of guesses that results in a hit from a given game state.

Note: We start in a state with no partially sunk ships. This is true as otherwise we will switch to the algorithms from Section 4 to sink the partially sunk ship and then come back to this problem. If all ships are sunk the game is over. Thus, the starting state for this algorithm never has any partially sunk ships.

3.2 Mathematical Formulation

3.2.1 Instance

- Grid size: $m \times n$ (section 2.1)
- Ship information: $shipLen$ of length n_s (section 2.2)

- Guess State Matrix for guessing player: $Guess_p$ (section 2.4)

Since the opponent's ship arrangement (Section 2.3) is unknown, it is not an input to our algorithm.

3.2.2 Solution

A sequence S of guesses made by player p: $[(x_1, y_1), (x_2, y_2)..(x_i, y_i)..(x_k, y_k)]$

3.2.3 Constraints

Guess $S[i] = (x_i, y_i)$ is a valid grid position, i.e. $1 \leq x_i \leq m, 1 \leq y_i \leq n$

All guesses are unique $\forall i, j \; i \neq j \implies S[i] \neq S[j]$

None of the guesses have already been made in the initial guess matrix, $\forall i, Guess_p[S[i]] = U$

The guesses in S are made in order by p

$Guess_p[S[i]] = M \; \forall 1 \leq i \leq k - 1$

$Guess_p[(x_k, y_k)] = H$. This is because otherwise we can find a solution of smaller size that finds the first hit.

3.2.4 Objective

minimize k

3.3 Solution 1: Random Guessing

In this algorithm we are going to take a naive random guessing approach. The intent here is to establish a baseline approach with some analysis. This would serve as a good comparison to the more complex solutions we come up with and establish their requirement if they perform better than this.

The algorithm is simply picking with uniform probability one of the locations on the grid which has not been guessed yet and guessing that. In case the guess results in a Hit we are done otherwise we pick another location. This process continues till we get a hit. Since, we do not know the arrangement of the opponent's ships we will assume there is a sub-routine, *checkGuess* that tells us for a given guess whether it is a hit or a miss. This sub-routine is the simulation of the opponent in the game telling you hit or miss for a guess.

3.3.1 Pseudocode

The pseudocode for the algorithm is shown in Algorithm 1

Algorithm 1 firstHitRandom

```
Input: m, number of rows in the grid
Input: n, number of columns in the grid
Input: Guess_p, Guess matrix of size m × n for the player
Output: guessSeq, Sequence of guesses to get the first hit
Available ← []                          ▷ Initialize the list of available guesses
guessSeq ← []                           ▷ Initialize the sequence of guesses to be made
for i = 1 to m do
    for j = 1 to n do
        if Guess_p[i, j] = U then
            Available.append((i, j))
        end if
    end for
end for
randomShuffle(Available)                ▷ Get a random permutation of the list using random shuffle.
for idx = 1 to len(Available) do
    G ← Available[idx]
    Guess_p[G] ← checkGuess(G)
    guessSeq.append(G)
    if Guess_p[G] = H then
        break
    end if
end for
return guessSeq
```

Implementation Details

Guess State Matrix for guessing player is stored as an array: $Guess_p$

Array storing available unexplored cells in the grid : *Available*.

Sequence of guesses to get the first hit : *guessSeq*

3.3.2 Proof of Correctness

The correctness for this algorithm is trivial. By definition we get all possible guesses that could be made and then construct a sequence of guesses where all guesses except the last are a miss. This gives a valid solution for finding the first hit.

3.3.3 Runtime Analysis

Initialization takes time $O(mn)$ as we visit every location to determine if it is available as a guess.

We cannot do better than this time by avoiding this step. Even if a very small number of guesses are left we do not know what they are unless that information is given to us beforehand.

Say number of available guesses is n_g, then random shuffle takes $O(n_g)$ time using Fisher-Yates Shuffle

We loop over this shuffled list of guesses till we find a hit. The operations within the loop are constant time so if the loop ran n_a times to find the first hit we take $O(n_a)$ time for this step.

The time complexity of this algorithm is $O(mn + n_g + n_a)$

Worst Case

In the worst case we would have all the guesses available which is starting from the initial state. Here, $n_g = mn$ so we can simplify the complexity to $O(mn + n_a)$.

Further, the number of possible squares ships can be on is $S = \sum_{i=1}^{n_s} shipLen[i]$

In the worst case we could guess every single location without a ship before the first hit thus needing $n_a = mn - S + 1$ guesses.

So, the worst case time in an adversarial setting is $O(mn + mn - S + 1) = O(mn)$

However, since we pick guesses randomly we are concerned with the expected time complexity. If we are not given the initial set of guesses the complexity will be $O(mn)$ which means we cannot improve. However, in an actual game we start with all guesses available and can remove the guess taken from the list as we go along so when the algorithm to find the first hit it called it can be given the list of available guesses as an input.

In this scenario the time complexity is $O(n_g + n_a)$.

Since, n_g is an input we can only analyze the time taken with respect to n_a.

Expected value of n_a

We can see that n_a is a random variable. In an initial state with n_g available guesses and n_{sp} unexplored locations containing a ship available, $(1 \leq n_{sp} \leq S)$,

$$Pr[n_a = k] = \frac{1}{k}\left(\frac{\binom{n_g - n_{sp}}{k-1}\binom{n_{sp}}{1}}{\binom{n_g}{k}}\right)$$

This is because we can choose $k - 1$ misses in $\binom{n_g - n_{sp}}{k-1}$ ways, 1 hit in $\binom{n_{sp}}{1}$ ways, and there are $\binom{n_g}{k}$ ways to select k guesses. Only $\frac{1}{k}$ of those arrangements have the hit at the end as the hit can equally likely be placed anywhere in the sequence.

Interesting Note: This is also known as the Hypergeometric distribution when computed over the number of successes rather than the number of attempts. However, we are fixing the number and location of the success and trying to find the expected value of the number of attempts required.

$$E[n_a] = \sum_{k=1}^{n_g - n_{sp} + 1} k \times Pr[n_a = k]$$

This expression is too complex to solve so we tried a different approach which was fix one parameter and evaluate the expected value on a range of the other and fit that data to a function to see if we could get the constants.

Fitting to n_g

We computed $E[n_a]$ for different values of n_g keeping n_{sp} constant. This experiment was repeated for different values of n_{sp}. We show in Figure 3 the outputs of this experiment for $n_{sp} = 30$.

We observed a linear relation between n_g and $E[n_a]$ for all experiments. So, we fit $E[n_a] = a(n_g + b)^c$. This gave a as different constants for different values of n_{sp}, $b = 1$ and $c = 1$ across all experiments.

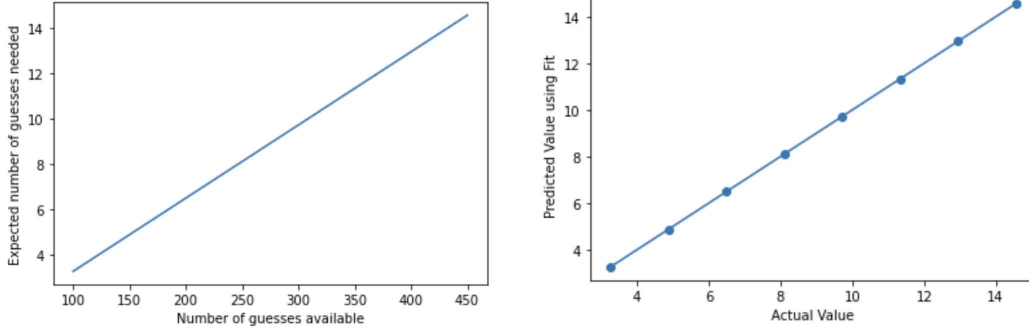

Figure 3: Left: The relation between n_g and $E[n_a]$ which we can see is linear. Right: The predicted and actual values of $E[n_a]$. This is a perfect fit which shows that the learned relation is correct.

Thus, we can say that $E[n_a] \propto n_g + 1$

Fitting to n_{sp}

Similarly, we computed $E[n_a]$ for different values of n_{sp} keeping n_g constant. This experiment was repeated for different values of n_g. We show in Figure 4 the outputs of this experiment for $n_g = 100$.

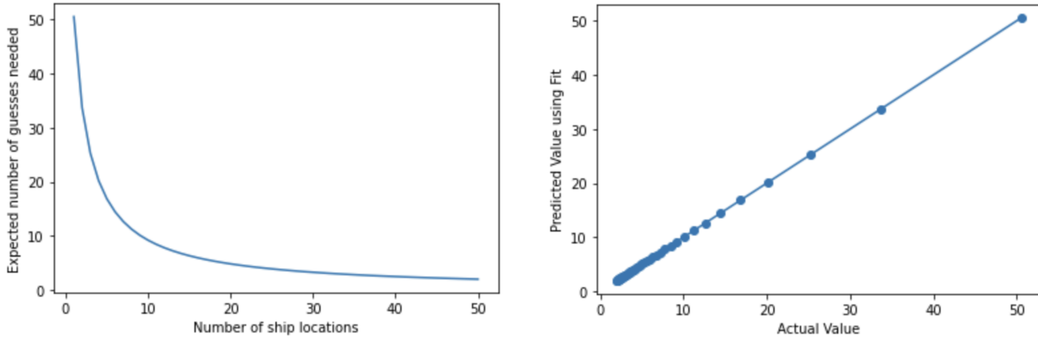

Figure 4: Left: The relation between n_{sp} and $E[n_a]$ which we can see is hyperbolic. Right: The predicted and actual values of $E[n_a]$. This is a perfect fit which shows that the learned relation is correct.

We observed a hyperbolic relation between n_{sp} and $E[n_a]$ for all experiments. So, we fit $E[n_a] = a(n_g + b)^{-c}$. This gave a as different constants for different values of n_g, $b = 1$ and $c = 1$ across all experiments. Thus, we can say that $E[n_a] \propto \frac{1}{n_{sp}+1}$

Further, in all our experiments we verified and it turns out the constants we got were nothing but the proportionality dependent on the other variable in each case. Combining these we got the result,

$$E[n_a] = \frac{n_g + 1}{n_{sp} + 1}$$

This, fit all the values we had tested. Additionally it satisfies sanity checks such as,

- $n_g = n_{sp}$, all guesses have ship locations: we get $E[n_a] = 1$ as we will get a hit for any guess
- $n_{sp} = 1$, 1 ship square is left: We get $E[n_a] = \frac{n_g+1}{2}$ as it is equally likely to find a hit with any of the guesses and this is just the average of all possible guess counts.

So, if we are given the list of available guesses as an input the expected runtime of the algorithm is

$$O\left(n_g + \frac{n_g + 1}{n_{sp} + 1}\right) = O(n_g)$$

3.4 Solution 2: Guessing Diagonally with parity

An interesting observation is that we do not have to go over all of the possible guesses to guarantee that we will find a hit. In trying to place a domino on a chess board we are guaranteed that at least 1 white square and at least 1 black square will be taken. So we have to guess through only 32 black or 32 white squares rather than all 64 squares. Similarly, we can get a set of guesses guaranteed to have 1 hit based on the size of the largest unsunk ship left.

These guesses would be along diagonals on the grid running going towards the right and down. We can start with the diagonal starting at $(1, 1)$. Say the length of the largest unsunk ship is m_s. The next diagonal by parity would be the one that starts at $(m_s + 1, 1)$ or $(1, m_s + 1)$ as these ensure that one positioning of the ship is covered only by one of our guesses. we continue guessing along such diagonals on either side of our starting diagonal till we run out of them or we find a hit.

The list of guesses to consider is, all unexplored main diagonal entries of the form i, j where $i = j$. Then check for diagonals along the row. If the first diagonal starts at $(1, 1)$ and $m_s = 6$, our first diagonal along the rows would start from $(7, 1)$, the second would start at $(13, 1)$ and so on, until i reaches m. Finally, add diagonals along the columns. Using the same logic as rows, if the first diagonal starts at $(1, 1)$ and $m_s = 6$, our first diagonal along the columns would start from $(1, 7)$, the second would start at $(1, 13)$ and so on, until j reaches n.

On analyzing this algorithm in simulations we identified that instead of going through all parity based guesses it was better to order them by the number of possible arrangements of the largest unsunk ship they can be a part of and guess in the decreasing order of this count. The data based intuition for doing this will be detailed in our Implementation Report. The mathematical intuition is that if our first guess is the one with the highest likelihood possible and so on for further guesses then the expected number of guesses required would achieve the lower bound for this approach. Thus, within the unexplored set of parity guesses, our algorithm selects the next move based on the highest likelihood of a hit.

The set of guesses on a board of dimensions 10×15 with $m_s = 6$ are shown in Figure 5.

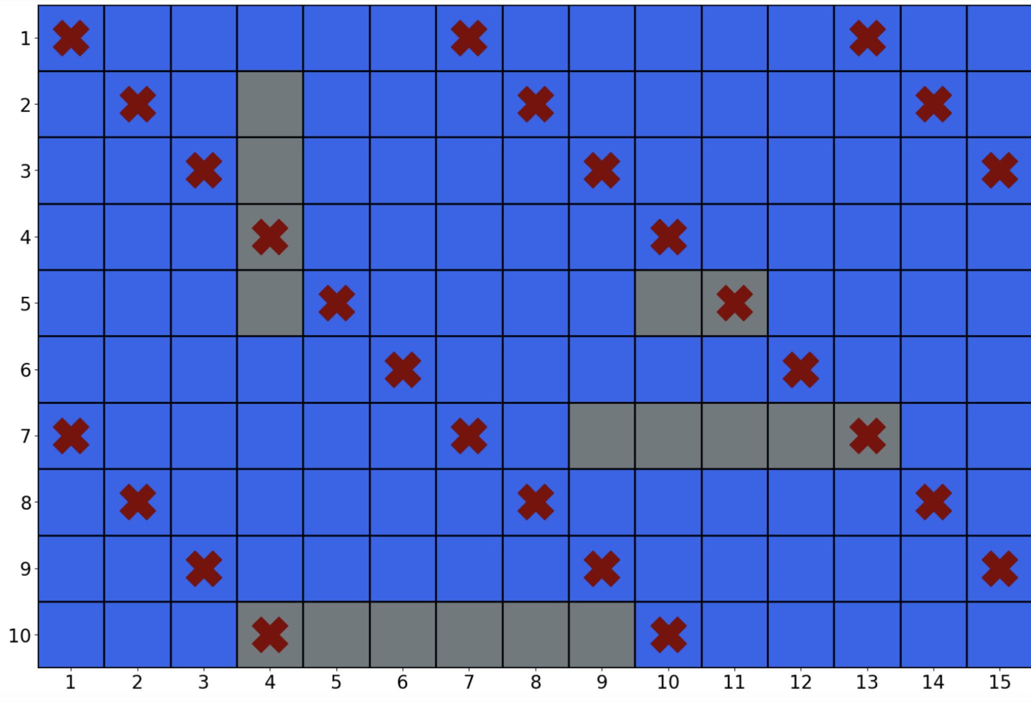

Figure 5: The set of diagonal parity guesses. $m = 10, n = 15, m_s = 6$

3.4.1 Pseudocode

The algorithm needs a subroutine that goes through all possible diagonal parity guesses and populates a hashmap with the number of arrangements of the largest unsunk ship possible for a cell as key and the list of guesses with that count of arrangements as value. We will then loop over the number of arrangements in decreasing order and make guesses till we find a hit.

The pseudocode for the subroutine is in Algorithm 4 and the algorithm is shown in Algorithm 5. We also have a couple of helper functions we have defined in Algorithm 2 and 3.

We will assume that we can get which ship is sunk and which is not, in constant time as that can be tracked as the game goes along. Thus, instead of *shipLen* we can take the longest unsunk ship length, m_s, as input. If we had started with a sorted list of ship lengths which we can in the initial state this would also be a constant time operation. So, these inputs can be assumed without any additional cost.

Algorithm 2 getDiagonalLocations

Input: m, number of rows in the grid
Input: n, number of columns in the grid
Input: i, x coordinate of first point in diagonal
Input: j, y coordinate of first point in diagonal
Output: *guessList*, List of guesses on the diagonal
$guessList \leftarrow []$
while $i \leq m$ and $j \leq n$ **do**
  $guessList.append((i, j))$
  $i \leftarrow i + 1$
  $j \leftarrow j + 1$
end while
return *guessList*

Algorithm 3 getAvailCount

Input: m, number of rows in the grid
Input: n, number of columns in the grid
Input: x, x coordinate of the point
Input: y, y coordinate of the point
Input: m_s, size of longest unsunk ship
Input: x_u, Directional update to make to x
Input: y_u, Directional update to make to y
Input: $Guess_p$, Guess matrix of size $m \times n$ for the player
Output: *availCount*, Number of continuous free spaces in that direction
$availCount \leftarrow 0$
$i \leftarrow x + x_u$
$j \leftarrow y + y_u$
while $max(0, x - m_s) < i < min(m + 1, x + m_s)$ and $max(0, y - m_s) < j < min(n + 1, y + m_s)$ **do**
  if $Guess_p[i, j] \neq U$ **then**
    break
  end if
  $availCount \leftarrow availCount + 1$
  $i \leftarrow i + x_u$
  $j \leftarrow j + y_u$
end while
return *availCount*

Implementation Details

Array of guesses on the diagonal : *guessList*

Guess matrix of size m × n for the player p : $Guess_p$

Array of guesses along the diagonals : *diagGuesses*

Algorithm 4 getDiagonalCounts

Input: m, number of rows in the grid
Input: n, number of columns in the grid
Input: m_s, Size of the largest unsunk ship
Input: $Guess_p$, Guess matrix of size $m \times n$ for the player
Output: $countMap$, Hashmap with key number of arrangements and value list of guesses

```
diagGuesses ← getDiagonalLocations(m, n, 1, 1)            ▷ Initialize with main diagonal
i ← m_s + 1
while i ≤ m do                                             ▷ Get all row wise diagonals
    diagGuesses ← diagGuesses + getDiagonalLocations(m, n, i, 1)
    i ← i + m_s
end while
j ← m_s + 1
while j ≤ n do                                             ▷ Get all column wise diagonals
    diagGuesses ← diagGuesses + getDiagonalLocations(m, n, 1, j)
    j ← j + m_s
end while
Initialize empty map countMap
for idx = 1 to 2m_s do
    countMap.put(idx, [])
end for
for idx = 1 to len(diagGuesses) do
    G ← diagGuesses[idx]
    if Guess_p[G] ≠ U then
        continue
    end if
    avail_a ← getAvailCount(m, n, G[0], G[1], m_s, −1, 0, Guess_p)
    avail_b ← getAvailCount(m, n, G[0], G[1], m_s, 1, 0, Guess_p)
    avail_l ← getAvailCount(m, n, G[0], G[1], m_s, 0, −1, Guess_p)
    avail_r ← getAvailCount(m, n, G[0], G[1], m_s, 0, 1, Guess_p)
    count ← max((avail_a + avail_b + 2 − m_s), 0) + max((avail_l + avail_r + 2 − m_s), 0)
    countMap.get(count).append(G)
end for
return countMap
```

Hashmap with key number of arrangements and value list of guesses : $countMap$, where amortized cost for *get* and *put* is $O(1)$.

Sequence of guesses to get the first hit : $guessSeq$

3.4.2 Proof of Correctness

Lemma 1: The number of distinct arrangements of the longest ship including cell i, j are $\max((avail_a + avail_b + 2 - m_s), 0) + \max((avail_l + avail_r + 2 - m_s), 0)$

$avail_a$ and $avail_b$ are the number of continuous unexplored cells above and below the cell i, j. Each count is limited to $m_s - 1$ since any cells beyond are not relevant in locating a ship of length m_s including i, j.

Since, i, j is also unexplored we get a continuous sequence of $avail_a + avail_b + 1$ unexplored cells. This means we can start at any of the first $avail_a + avail_b + 1 - (m_s - 1)$ cells and have the ship fit within this stretch and contain i, j.

Thus, total number of vertical arrangements possible are $avail_a + avail_b + 2 - m_s$. We need to take a maximum with 0 however in case this expression is negative which is possible. The maximum possible is m_s as both $0 \leq avail_a, avail_b \leq m_s - 1$. Similarly, the number of horizontal arrangements possible are $avail_l + avail_r + 2 - m_s$. This also needs to have a maximum taken with 0 to cover for cases in

Algorithm 5 firstHitDiagonal

Input: m, number of rows in the grid
Input: n, number of columns in the grid
Input: m_s, Size of the largest unsunk ship
Input: $Guess_p$, Guess matrix of size $m \times n$ for the player
Output: $guessSeq$, Sequence of guesses to get the first hit

```
availableMap ← getDiagonalCounts(m, n, m_s, Guess_p)   ▷ Initialize the map counts with available guesses
guessSeq ← []                                           ▷ Initialize the sequence of guesses to be made
for count = 2m_s to 1 do
    Available ← availableMap.get(count)
    for i = 1 to len(Available) do
        G ← Available[idx]
        Guess_p[G] ← checkGuess(G)
        guessSeq.append(G)
        if Guess_p[G] = H then
            break
        end if
    end for
end for
return guessSeq
```

which the expression is negative. Thus, total number arrangements are $\max((avail_a + avail_b + 2 - m_s), 0) + \max((avail_l + avail_r + 2 - m_s), 0)$, which can be a maximum of $2m_s$

Lemma 2: Any continuous sequence of length m_s contains exactly 1 cell from *diagGuesses*

We will prove this by contradiction. Assume there is a sequence $A = (i_1, j_1), \ldots, (i_{m_s}, j_{m_s})$ such that $(i_k, j_k) \notin diagGuesses, \forall k$.

If A is horizontally aligned $i_1 = i_2 = \ldots = i_{m_s}$. Then, $j_1 = j_2 - 1 = j_3 - 2 = \ldots = j_{m_s} - (m_s - 1)$

Given the construction of our diagonals, for some guess at i_1, y, we also have a guess at $i_1, y + m_s$ assuming it is in bounds. There are $m_s - 1$ cells in between. If A does not include either of the two then it must lie between the two. However, A has m_s continuous locations meaning that cannot happen. Additionally, it can only contain one of the two as to contain both the length would need to be $> m_s$.

Similar argument can be made for a vertical orientation. Thus, we show that any continuous sequence of m_s contains exactly 1 cell from *diagGuesses*.

Using Lemma 2, we can say that we are guaranteed to find a hit on exhausting all guesses from *diagGuesses* as one of them has to contain a cell from the ship of size m_s. Additionally, this set of guesses is a minimal set within an additive error of constant cells with this property as each arrangement contains only 1 such cell which means possible arrangements for each cell are mutually exclusive.

3.4.3 Runtime Analysis

The length of the array *diagGuesses* can be computed as follows,

The main diagonal contains m cells, 1 for each row. (Recall $m \leq n$)

There are $floor(\frac{m-1}{m_s})$ diagonals row wise of length $m - m_s$, $m - 2m_s$, ..., one cell per row. This is $O(\frac{m^2}{m_s})$ guesses.

There are $floor(\frac{n-1}{m_s})$ diagonals column wise of length $min(n - m_s, m)$, $min(n - 2m_s, m)$, ..., one cell per column, unless it gets bounded by the number of rows. This is $O(\frac{mn}{m_s})$ guesses.

We have $O(\frac{mn}{m_s} + \frac{m^2}{m_s} + m)$ guesses. $O(m)$ is dominated as $m_s \leq m$ and $n \geq m$. Thus, the total number

of guesses are

$$O\left(\frac{mn}{m_s}\right)$$

Figure 6 illustrates the number of available arrangements for a ship of length 6 on a board of 10×15 along the diagonal parity guesses.

This is better than the $O(mn)$ candidate guesses that the random algorithm had.

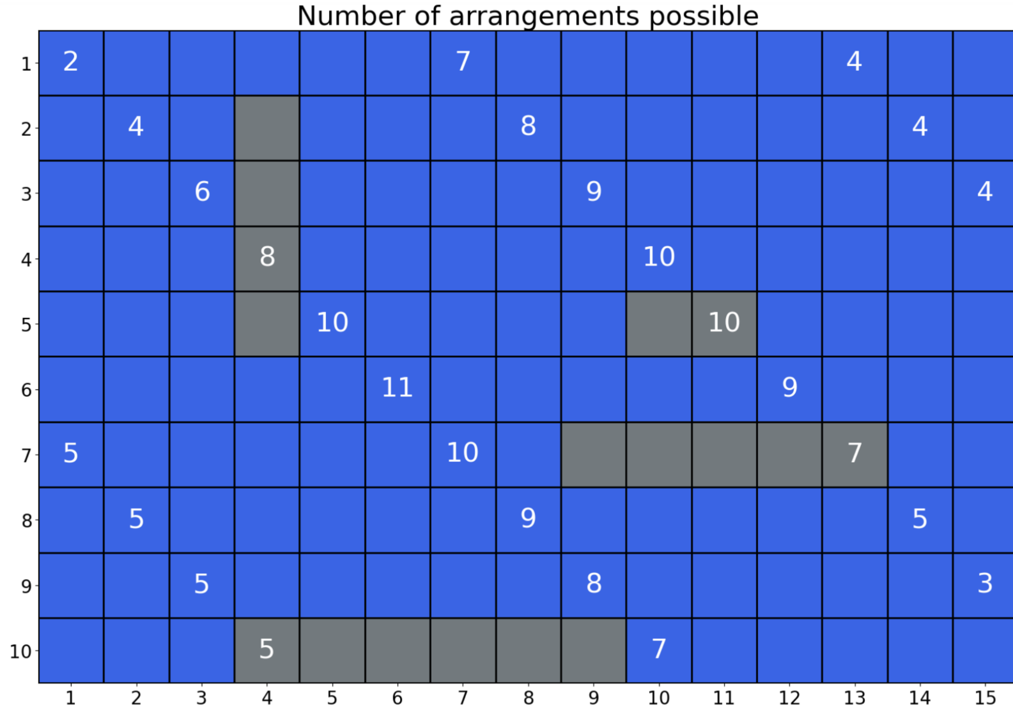

Figure 6: The count of possible arrangements for each parity guess for $m = 10, n = 15, m_s = 6$. Notice the top left corner has 2 as it can only be a part of 1 horizontal and 1 vertical arrangement.

Overall Complexity

Getting the list of all guesses is iterating over each guess once and no extra cells so $O\left(\frac{mn}{m_s}\right)$

To compute the hashmap, we go over every guess and explore upto $m_s - 1$ cells in 4 directions to check for avail. Then, in $O(1)$ time we compute the number of arrangements for the cell and in $O(1)$ time add the guess to the hashmap. The complexity to prepare the hashmap is $O\left(\frac{mn}{m_s} \times m_s\right) = O(mn)$

Then we loop over every possible count and get the list of guesses with that count. The inner loop checks each guess. These loops overall check every guess once at most. The check takes $O(1)$ time. The time complexity of this step is $O(n_a)$ which is the number of attempts needed to get the first hit.

Combining, the complexity is $O(mn)$ as the other two terms are dominated by this.

Comparing to Random

The overall complexity of this algorithm is $O(mn)$ which we cannot do better on.

However, the expected number of guesses needed here can be compared to the expected number of guesses needed by the random algorithm.

Recall, $E[n_a] = \frac{n_g+1}{n_{sp}+1}$ was the expected number of guesses for the random algorithm.

Computing $E[n_a] = \sum_{k=1}^{\frac{mn}{m_s}} k \times Pr[n_a = k]$ for the diagonal algorithm we can note that $Pr[n_a = k]$ is non-increasing in k. This is because we first guess the locations with highest possible arrangements. This ensures that the expected value is minimum possible for this set of guesses.

The diagonal guess is a subset of the random guessing in any scenario with a guaranteed hit so in most cases it will give a faster hit. However, in certain specific settings, the random algorithm may give the result in a smaller number of guesses as this algorithm does not account for any ship other than the largest unsunk. On the other hand if most guesses in the set *diagGuesses* have been made then this algorithm is a very quick path to finding the first hit.

We will further analyze the choice using our simulation results in the implementation report.

3.5 Solution 3: Guessing with PDF

The Diagonal guessing while structured and reduces the number of guesses, was limited. It is hard to adapt for scenarios where we might be starting somewhere in the middle of the game. It also views the board considering only the largest unsunk ship which does not account for the information about where the remaining ships would be.

The intuition for this algorithm is based in the idea of the diagonal guessing. We can calculate the number of arrangements of the largest ship that goes through any cell on the board. This can be done for every ship and we can add them to get a number representing the total possible ship placements including the cell. This would be a Probability Density Function (PDF) of the likelihood of a cell being occupied by a ship.

We guess in decreasing order of this likelihood till we find a hit. In the diagonal guessing, a miss did not require any update as the arrangements being ruled out were mutually exclusive. This is not true in this case. So, we will update the vertical and horizontal neighbors by reducing the number of arrangements that are eliminated when we find a miss.

3.5.1 Pseudocode

We need subroutines to generate and update the PDF shown in Algorithm 6 and 7, illustrated in Figure 7 and 8 respectively. We also use Algorithm 3. The algorithm is shown in Algorithm 8.

We can assume every time a ship is sunk it is removed from all lists for use in the algorithm. This would have amortized constant time cost. So, we will assume that the array $shipLenUS$ contains the lengths of unsunk ships in increasing order and can be used as input.

Number of arrangements possible

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	20	30	36	42	45	46	46	46	46	46	45	42	36	30	20
2	30	40	46	52	55	56	56	56	56	56	55	52	46	40	30
3	36	46	52	58	61	62	62	62	62	62	61	58	52	46	36
4	42	52	58	64	67	68	68	68	68	68	67	64	58	52	42
5	45	55	61	67	70	71	71	71	71	71	70	67	61	55	45
6	45	55	61	67	70	71	71	71	71	71	70	67	61	55	45
7	42	52	58	64	67	68	68	68	68	68	67	64	58	52	42
8	36	46	52	58	61	62	62	62	62	62	61	58	52	46	36
9	30	40	46	52	55	56	56	56	56	56	55	52	46	40	30
10	20	30	36	42	45	46	46	46	46	46	45	42	36	30	20

Figure 7: The count of possible arrangements for $m = 10, n = 15, shipLen = [2, 4, 5, 6]$.

Algorithm 6 getPDF

Input: m, number of rows in the grid
Input: n, number of columns in the grid
Input: $shipLenUS$, Array containing lengths of unsunk ships in sorted order
Input: $Guess_p$, Guess matrix of size $m \times n$ for the player
Output: $pdfMatrix$, Matrix of size $m \times n$ containing the PDF
Initialize $pdfMatrix$ empty array of size $m \times n$
$m_s \leftarrow shipLenUS[length(shipLenUS)]$ ▷ Maximum unsunk ship length as the array is sorted
for i = 1 to m: **do**
  for j = 1 to n: **do**
    $currCount \leftarrow 0$
    if $Guess_p[i, j] = U$ **then**
      $a \leftarrow getAvailCount(m, n, i, j, m_s, -1, 0, Guess_p)$
      $b \leftarrow getAvailCount(m, n, i, j, m_s, 1, 0, Guess_p)$
      $l \leftarrow getAvailCount(m, n, i, j, m_s, 0, -1, Guess_p)$
      $r \leftarrow getAvailCount(m, n, i, j, m_s, 0, 1, Guess_p)$
      for k = 1 to $length(shipLenUS)$: **do**
        $s \leftarrow shipLenUS[k] - 1$
        $currCount \leftarrow currCount + \max(min(a, s) + min(b, s) + 2 - (s + 1), 0)$
        $currCount \leftarrow currCount + \max(min(l, s) + min(r, s) + 2 - (s + 1), 0)$
      end for
    end if
    $pdfMatrix[i, j] \leftarrow currCount$
  end for
end for
return $pdfMatrix$

Number of arrangements possible

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	20	30	36	42	45	46	46	45	46	46	45	42	36	30	20
2	30	40	46	52	55	56	56	52	56	56	55	52	46	40	30
3	36	46	52	58	61	62	62	52	62	62	61	58	52	46	36
4	42	52	58	64	67	68	68	52	68	68	67	64	58	52	42
5	45	55	61	67	70	71	71	45	71	71	70	67	61	55	45
6	45	55	60	63	60	55	45		45	55	60	63	60	55	45
7	42	52	58	64	67	68	68	42	68	68	67	64	58	52	42
8	36	46	52	58	61	62	62	46	62	62	61	58	52	46	36
9	30	40	46	52	55	56	56	46	56	56	55	52	46	40	30
10	20	30	36	42	45	46	46	42	46	46	45	42	36	30	20

Figure 8: Recomputed PDF after miss at (6,8).

Implementation details

We use arrays and matrices for variables as appropriate.

Matrix of size m $\times$ n containing the PDF : $pdfMatrix$

Algorithm 7 updatePDF

```
Input: m, number of rows in the grid
Input: n, number of columns in the grid
Input: x, x coordinate of miss
Input: y, y coordinate of miss
Input: shipLenUS, Array containing lengths of unsunk ships in sorted order
Input: Guess_p, Guess matrix of size m × n for the player
Input: pdfMatrixIn, Input PDF matrix
Output: pdfMatrixOut, Updated PDF matrix
pdfMatrixOut ← pdfMatrixIn
m_s ← shipLenUS[length(shipLenUS)]      ▷ Maximum unsunk ship length as the array is sorted
j ← y
for i = max(x − m_s + 1, 1) to min(x + m_s − 1, m): do
    currCount ← 0
    if Guess_p[i, j] = U then
        a ← getAvailCount(m, n, i, j, m_s, −1, 0, Guess_p)
        b ← getAvailCount(m, n, i, j, m_s, 1, 0, Guess_p)
        l ← getAvailCount(m, n, i, j, m_s, 0, −1, Guess_p)
        r ← getAvailCount(m, n, i, j, m_s, 0, 1, Guess_p)
        for k = 1 to length(shipLenUS): do
            s ← shipLenUS[k] − 1
            currCount ← currCount + max(min(a, s) + min(b, s) + 2 − (s + 1), 0)
            currCount ← currCount + max(min(l, s) + min(r, s) + 2 − (s + 1), 0)
        end for
    end if
    pdfMatrixOut[i, j] ← currCount
end for
i ← x
for j = max(y − m_s + 1, 1) to min(y + m_s − 1, n): do
    currCount ← 0
    a ← getAvailCount(m, n, i, j, m_s, −1, 0, Guess_p)
    b ← getAvailCount(m, n, i, j, m_s, 1, 0, Guess_p)
    l ← getAvailCount(m, n, i, j, m_s, 0, −1, Guess_p)
    r ← getAvailCount(m, n, i, j, m_s, 0, 1, Guess_p)
    for k = 1 to length(shipLenUS): do
        s ← shipLenUS[k] − 1
        currCount ← currCount + max(min(a, s) + min(b, s) + 2 − (s + 1), 0)
        currCount ← currCount + max(min(l, s) + min(r, s) + 2 − (s + 1), 0)
    end for
    pdfMatrixOut[i, j] ← currCount
end for
return pdfMatrixOut
```

Guess matrix of size m $\times$ n for the player : $Guess_p$

We use a max-heap to implement $pdfHeap$ which is of size which takes $O(\log n)$ time in deletion and insertion.

It takes time $O(n)$ to construct using the divide and conquer buildHeap algorithm. Here n is the size of the heap.

3.5.2 Proof of Correctness

The correctness for this algorithm is trivial. By definition we get all possible guesses that could be made and then construct a sequence of guesses where all guesses except the last are a miss. This gives a valid solution for finding the first hit.

Algorithm 8 firstHitPDF

```
Input: m, number of rows in the grid
Input: n, number of columns in the grid
Input: shipLenUS, Array containing lengths of unsunk ships in sorted order
Input: Guess_p, Guess matrix of size m × n for the player
Output: guessSeq, Sequence of guesses to get the first hit
m_s ← shipLenUS[length(shipLenUS)]        ▷ Maximum unsunk ship length as the array is sorted
pdfMatrix ← getPDF(m, n, shipLenUS, Guess_p)
guessSeq ← []
Initialize maxHeap pdfHeap ← Heapify(pdfMatrix)        ▷ i, j ordered by pdfMatrix[i, j] > 0
while pdfHeap is not empty do
    G ← pdfHeap.extractMax()        ▷ This also deletes from the heap
    Guess_p[G] ← checkGuess(G)
    guessSeq.append(G)
    if Guess_p[G] = H then
        break
    end if
    pdfMatrix ← updatePDF(m, n, G[0], G[1], shipLenUS, Guess_p, pdfMatrix)
    j ← G[1]
    for i = max(G[0] − m_s + 1, 1) to min(G[0] + m_s − 1, m): do
        pdfHeap.delete((i, j))
        pdfHeap.insert((i, j), pdfMatrix[i, j])    ▷ Updating the heap to reflect the new PDF values
    end for
    i ← G[0]
    for j = max(G[1] − m_s + 1, 1) to min(G[1] + m_s − 1, n): do
        pdfHeap.delete((i, j))
        pdfHeap.insert((i, j), pdfMatrix[i, j])    ▷ Updating the heap to reflect the new PDF values
    end for
end while
return guessSeq
```

3.5.3 Runtime Analysis

Generation of the PDF goes through every cell. These are $O(mn)$. The inner loop process is only run for n_g, number of unexplored cells.

To compute the pdf for a cell we compute the availability in each direction which take $O(m_s)$ time.

Then we loop through each ship and take constant time to add number of possible arrangements which takes $O(n_s)$ time where n_s is the number of unsunk ships.

Combining we can say that PDF generation takes $O(mn + n_g(m_s + n_s))$ time.

Say we go through n_a attempts to get the first hit.

For each of the first $n_a - 1$ attempts we update the PDF. We update $O(m_s)$ neighbors of the cell that was a miss and each update takes time $O(m_s + n_s)$ as in PDF generation. So time taken to update PDF across all guesses is $O(n_a m_s(m_s + n_s))$

Note: Ideally we only need to update the avail in one direction and recompute the pdf for each cell however that would be asymptotically similar. In an actual implementation we can save time by maintaining *avail* arrays for all 4 directions. This will be followed in the implementation phase.

Generating the heap initially takes $O(n_g)$ time. In each loop we extract the max which takes $O(1)$ time but since that happens with deletion the time is $O(\log n_g)$. Then we perform $O(m_s)$ inserts and deletes in the heap. Other operations are updating the pdf which we already counted or constant time.

The loop takes time $O(n_a m_s \log n_g)$. Thus, the total time taken is $O(mn + n_g(m_s + n_s) + n_a m_s(m_s + n_s + \log n_g))$. Further, simplification needs formulation of the different variables.

Note that the $E[n_a]$ is the minimum possible using this algorithm. In any strategy we have to generate

an ordering of all possible guesses and that determines the expected value. This algorithm has all possible guesses and orders them in the order of non-increasing probabilities. Thus, ensuring that when computing $E[n_a]$ there is no possible switch in the sequence of guesses that will further decrease it. This is because higher probability at lower values of n_a means the expectation will be lower.

Thus, we can claim that over expectation this will be the best strategy. We will see further empirical analysis on this in the implementation report.

3.6 Discussion

We discussed 3 different solutions to finding the first hit in this section. We saw that solution 3 was the best and solution 2 was better than solution 1 in expected time. However, we must note that a better quality solution costs us in time complexity and requires tracking of more complex information. In a real world setting a human player may not be able to compute and maintain a pdf. Realistically the suggested solution for most people would be solution 2 however if you can, play using solution 3.

4 Sink the Ship

4.1 Problem Statement

Given a state in which a player p has a successful hit we would like to sink the ship that is hit in the minimum number of guesses.

This problem has 2 variants one in which player q will provide the length of the ship that has been hit and the other in which this information is not shared with player p. We will refer to the variant with the length information as Variant 1 and without the information as Variant 2.

4.2 Mathematical formulation

4.2.1 Instance

- Grid size: $m \times n$ (section 2.1)
- Guess State Matrix for guessing player: $Guess_p$ (section 2.4)
- Ship information: $shipLen$ of length n_s (section 2.2)
- Identifier of ship that is hit: i [Input for variant 1 only]

Since the opponent's ship arrangement (Section 2.3) is unknown, it is not an input to our algorithm.

4.2.2 Solution

A sequence S of guesses made by player p: $[(x_1, y_1), (x_2, y_2)..(x_i, y_i)..(x_k, y_k)]$

4.2.3 Constraints

Guess $S[i] = (x_i, y_i)$ is a valid grid position, i.e. $1 \leq x_i \leq m, 1 \leq y_i \leq n$

All guesses are unique $\forall i, j \ i \neq j \implies S[i] \neq S[j]$

None of the guesses have already been made in the initial guess matrix, $\forall i, Guess_p[S[i]] = U$

The guesses in S are made in order by p

$$\forall (x, y),\ shipS_q[i] \leq (x, y) \leq shipE_q[i] \implies \exists j S[j] = (x, y)$$

In other words guesses should sink the ship. In Variant 2, i refers to the ship that was hit but is unknown to player p.

Also, $shipS_q[i] \leq S[k] \leq shipE_q[i]$, the final guess should be a hit on the ship which we are trying to sink. Combined with the previous constraint this is also the hit that sinks the ship. This is because otherwise we can find a solution of smaller size that sinks the ship.

4.2.4 Objective

minimize k

Note: In all these algorithms we run the risk of finding an arrangement in which different ships are arranged next to each other such that they might give consecutive hits in a direction without being the ship we first hit and wanted to sink. This is simply resolved by assuming that there is an additional declaration provided to us when a ship is sunk. So say if we are looking to sink a ship of length 3 after the first hit and we find 2 more hits to the left but no sunk announcement, this means they belonged to different ships and thus do not count towards our checks to determine the ship is sunk and we will try in a different direction. These hits on parallel ships will be noted and once we have completed sinking this ship the algorithm will be called on each one of them giving these as the first hit. To this end we assume a function, *hasSunkShip*, which takes no arguments and returns true if any new ship has been sunk. This simluates the player announcement that the ship is sunk.

4.3 Brute Force Algorithm

This variant of the solution makes ordered guesses of where the next hit location could be based on adjacent, valid guesses available. This works for both variants as in case we do not know the length of the ship that is hit we can assume the length to be the maximum unsunk ship length.

Let's say that player p found a hit at position (x, y) on player q's board. The index of the ship which has been hit is i in the shipLen array. The length of the ship which has been hit (shipLen[i]) is also given in this variant. Let H_i track the total number of hits discovered for this ship so far.

We first present the pseudocode[9] for this algorithm and then provide the explanation for it.

Algorithm Description

- We loop through an array of directions to find a hit in an adjacent unexplored cell.
- Once we find it we continue guessing in that direction and then in the opposite direction till we have sunk the ship or hit squares if the ship length
- We would need to call this function twice if it returns without a ship being sunk
- On the second call the direction explored would be perpendicular to the first as the first would not be unexplored anymore.

Time Analysis

- Worst case, for each ship we explore 4 directions.
- Each direction can give us an initial hit and an immediate miss except the correct one. This is constant time complexity.
- The loop entered is thus linearly dependent on the length of the ship $O(ShipLen[i])$

Proof of Correctness

The correctness proof of this algorithm is trivial. For the below constraints,

- Length of each ship in the play is at least 1. This provides a guarantee that all the ships have at least one successful hit coordinate.

Algorithm 9 Brute Force Algorithm

Input: m, number of rows in the grid
Input: n, number of columns in the grid
Input: fhx, fhy, grid coordinates of discovered first hit
Input: len_s, Length of the ship to be sunk
Input: $Guess_p$, Guess matrix of size $m \times n$ for the player
Output: $guessSeq$, Sequence of guesses to sink the ship

```
guessSeq ← []
hitCnt ← 1
directions ← [(−1, 0), (1, 0), (0, −1), (0, 1)]
for i = 1 to 4: do
    dir ← directions[i]
    G ← (fhx + dir[0], fhy + dir[1])
    if 1 ≤ G[0] ≤ m && 1 ≤ G[1] ≤ n && Guess_p[G] = U then
        Guess_p[G] ← checkGuess(G)
        guessSeq.append(G)
        if Guess_p[G] = H: then
            hitCnt ← hitCnt + 1
            break
        end if
    end if
end for
opposite ← False
while hitCnt < len_s||!hasSunkShip(): do
    G ← G + dir
    if 1 ≤ G[0] ≤ m && 1 ≤ G[1] ≤ n && Guess_p[G] = U then
        Guess_p[G] ← checkGuess(G)
        guessSeq.append(G)
        if Guess_p[G] = H then
            hitCnt ← hitCnt + 1
            G ← G + dir
        else
            if opposite then
                break
            else
                opposite ← True
                G ← (fhx, fhy)
                dir ← −1 ∗ dir
            end if
        end if
    end if
end while
return guessSeq
```

- There are no ships outside of the provided grid. That is search space is finite and the algorithm guarantees termination.
- Since the beginning of the play, the ships won't move. This guarantees that all possible hit coordinates are fixed and are bound to be found in constant time relative to the first-hit coordinate.

Now, since the ships are contiguous once we have the first hit it can only be located to the left and right or to up and down of it. So we are guaranteed that at least one of the directions will be a hit. Once that is a hit we continue going in that direction till we get a miss or the board ends. Then we go in the opposite direction. In case we sink the ship we are done otherwise these hits were other ships adjacent to the one we are trying to sink. Now, the ship is guaranteed to have perpendicular alignment. We call the function again and it goes in that direction to find and sink the ship. It will select the new direction as the previous one is already explored. Thus, our sink ship algorithm is guaranteed to hit all the coordinates where the ship has a presence.

If we find any hits that did not belong to the ship we sank we can call the sink the ship algorithm again on them as they are first hits.

4.4 Smart Algorithm

This variant of the solution tries not only to sink the ship found by the first hit phase of our complete solution but also tries to help the search phase for the next iteration. An explanation of the latter will follow in subsequent sections.

We first present the pseudocode[10] for this algorithm and then provide the explanation for it.

Algorithm 10 smartSinkShipV1

Input: m, number of rows in the grid
Input: n, number of columns in the grid
Input: fhx, fhy, grid coordinates of discovered first hit
Input: $shipLen$, array of length n_s with ship lengths on the grid sorted by length
Input: $Guess_p$, Guess matrix of size $m \times n$ for the player
Output: $guessSeq$, Sequence of guesses to sink the ship
Output: $anotherShipFound$, Sequence of guesses found for other ships

```
len_smallest = shipLen[0]          ▷ pick the smallest ship length from shipLen
anotherShipFound = []              ▷ If other ships are discovered, we return the coordinates
for i = 1 to 4 do                  ▷ i represents the direction, [(1, Up)(2, Down)(3, Left)(4, Right)]
    guesses, anotherShip = initiateHitSequence(m, n, Guess_p, fhx, fhy, i, len_smallest)
    guessSeq ← guessSeq + guesses
    anotherShipFound ← anotherShipFound + anotherShip
end for
return guessSeq, anotherShipFound
```

The smartSinkShipV1, Algorithm 10, uses two subroutines, *initiateHitSequence*, Algorithm 12, and *destroyTillMiss*, Algorithm 11. The pseudocode for both these subroutines is provided below.

Algorithm Description

- We start our sink ship phase with inputs like coordinates of the first hit, an array with ship information, the length of the biggest unsunk ship and a guess grid of the player.
- First we need to decide which direction relative to the first hit coordinate our ship may be placed.
- There is no better way than hitting a coordinate in each of the four directions to find the ultimate path to sink the ship.
- But we need to help the find first hit phase with this deduction. How do we do that?
- For a given first-hit coordinate, we are guaranteed to at least destroy the smallest unsunk ship of the opponent. We want to use this information to make our process better.

- We decide to hit the coordinates which are $len_{smallest}$ distance away from the first hit coordinate. If we get a hit, bingo! We have found our direction.
- This logic seems to miss corner cases like what if the ship is placed on both negative and positive axes taking the first hit coordinate as the origin. Therefore, we modify our logic to hit $\frac{len_{smallest}}{2}$ in each of the four possible directions. This logic is implemented in pseudocode[12].
- If we encounter a miss we hit coordinates in the opposite direction of the last step starting from the first hit coordinate till we encounter a miss/already explored coordinate.
- Another corner case for this approach is the unexplored coordinates between the first hit and $firstHitCo + \frac{len_{smallest}}{2}$. We may get a hit on $firstHitCo + \frac{len_{smallest}}{2}$ and then get a miss on $firstHitCo + 1$ coordinate on the decided direction.
- This means that we have discovered another ship in the same direction and must use this information to **trigger another sink ship phase without returning back to search first hit phase**. For the current sink ship, we keep on exploring other directions.
- As we can see, we assume that we are going to sink the smallest possible ship in this iteration but we may end up destroying a bigger one. We need to update this information into our ship arrays. This can be done separately as a clean up after every ship is sunk. This is amortized constant time operation compared to all the guesses we make and helps us maintain the ship arrays as only the unsunk ships with lengths sorted in increasing order.

Algorithm 11 destroyTillMiss

Input: m, number of rows in the grid
Input: n, number of columns in the grid
Input: fhx, fhy, grid coordinates of discovered first hit
Input: dir, direction of the hit sequence.
Input: $Guess_p$, Guess matrix of size $m \times n$ for the player
Output: $foundParts$, Boolean indicating if any hits were found
Output: $guessSeq$, Sequence of guesses to sink the ship
$guessSeq \leftarrow []$
$hitCnt \leftarrow 0$
$G \leftarrow (fhx + dir[0], fhy + dir[1])$
while $1 \leq G[0] \leq m$ && $1 \leq G[1] \leq n$ && $Guess_p[G] = U$ **do**
    $guessSeq.append(G)$
    $Guess_p[G] \leftarrow checkGuess(G)$
    if $Guess_p[G] = H$ **then**
        $hitCnt \leftarrow hitCnt + 1$
        $G \leftarrow G + dir$
    else
        break
    end if
end while
return $hitCnt > 0, guessSeq$

Time Analysis

- For each ship we explore 4 directions.
- Each direction can give us an an immediate miss except the correct ones. This means $4 + 3 * 2 + len_s + 1$ guesses are needed where len_s is the length of the ship sunk.
- Notice that we do not take a ship length as input so this algorithm can in the worst case run till the maximum unsunk ship length say m_s.
- Additionally we can have adjacent ships from the first hit point which will both be sunk by this algorithm doubling the number of guesses.

Algorithm 12 initiateHitSequence

Input: m, number of rows in the grid
Input: n, number of columns in the grid
Input: fhx, fhy, grid coordinates of discovered first hit
Input: i, direction of the hit sequence.
Input: $Guess_p$, Guess matrix of size $m \times n$ for the player
Input: $len_{smallest}$, smallest ship available in $shipLen$
Output: $guessSeq$, Sequence of guesses to sink the ship
Output: $anotherShip$, Sequence of guesses found for other ships
$anotherShip \leftarrow []$
$guessSeq \leftarrow []$
$directions \leftarrow [(-1,0),(1,0),(0,-1),(0,1)]$
$baseDir \leftarrow directions[i]$ ▷ directions is indexed by 1
$dirDelta \leftarrow baseDir * (len_{smallest}//2)$
$G \leftarrow (fhx + dirDelta[0], fhy + dirDelta[1])$
$guessStatus \leftarrow checkGuess(G)$
if $guessStatus = H$ **then**
  $foundParts, guessArr \leftarrow destroyTillMiss(m, n, fhx, fhy, baseDir, Guess_p)$
  $guessSeq \leftarrow guessSeq + guessArr$
  if $!foundParts$ **then**
    $anotherShip.append(G)$
  else
    $newDir \leftarrow -1 * baseDir$
    $foundParts, guessArr \leftarrow destroyTillMiss(m, n, fhx, fhy, newDir, Guess_p)$
    $guessSeq \leftarrow guessSeq + guessArr$
  end if
end if
return $guessSeq, anotherShip$

- Thus the number of guesses made by the algorithm are $O(m_s)$.
- Each guess requires constant time operations such as checking adding to matrix and then preparing the next guess.
- So, overall time complexity of the algorithm is $O(m_s)$. Notice that this is higher than the brute force
- The hope is that this cost pays of in discovering more nearby ships and being able to sink adjancent ships in one go.

Proof of Correctness

The correctness proof of this algorithm is similar to the brute force and trivial. For the below constraints,

- Length of each ship in the play is at least 1. This provides a guarantee that all the ships have at least one successful hit coordinate.
- There are no ships outside of the provided grid. That is search space is finite and the algorithm guarantees termination.
- Since the beginning of the play, the ships won't move. This guarantees that all possible hit coordinates are fixed and are bound to be found in constant time relative to the first-hit coordinate.

Since, the ship is contiguous and we go through every direction from the first hit till we find a miss if that direction or the opposite has a hit we are searching through the complete possibility space. Once we make all these guesses our sink ship algorithm is guaranteed to hit all the coordinates where the ship has a presence. Thus completing the sink process successfully.

4.5 Discussion

We discussed 2 different solutions to sinking the ship in this section. Both are similar at least as per asymptotic analysis with regards to the runtime and the number of guesses needed. Brute force sink is more efficient in the practical sense but it does need the ship length. In case of variant 2 it will be similar to smart sink in the number of guesses as well. Even though smart sink is slightly more expensive we are curious to evaluate it in implementation of a complete end to end game to see if that cost can be offset by it finding new ships much more often. More on this in the implementation report.

5 Conclusion

We saw algorithms to find the first hit in a small number of guesses and then sinking that ship. These work for any grid shape and any number of ships of different lengths. These can also be used starting at any state of the game rather than just the start which would allow switching between different algorithms possible. In case you choose to play a complete game with these, any of the find the first hit algorithms can be used to find a ship. Once we do, any of the sink the ship algorithms can be used to sink the ship and we continue in that cycle. Playing a complete game with these would also reduce the time complexity of the algorithms as we can initialize the variables needed at the start and update them as we go. Such as the PDF can be generated only once at the start and updated with gameplay thus removing that cost from the algorithm. Similarly the list of available guesses can be maintained and updated as we go along so that recomputation is not required. We intend to put different combinations together to run simulated games and analyze their performance for our implementation report. However, based on the analysis we believe these are close to the best algorithms for playing the game of battleship.