Task 1.

***Algorithm: RandomStrategy***

**Input**: board[0,1,...,n−1][0,1,...,n−1], set of remaining ships, Sr ⊆ S, set of miss coordinates, M, set of hit coordinates, H.

**Output**: fire the point of the board

**procedure** RandomStrategy(board, Sr, M, H)

x←0, y←0

**while** M.contains(x,y) == False and H.contains(x,y) == False

x ← random(0, n-1)

y ← random(0, n-1)

**end** **while**

return fire(board[y][x])

**end** **procedure**

The random function has constant-time complexity.

The time complexity of this algorithm is **O(n2 )** in the worst-case scenario, as it may iterate over all n2 positions on the board before finding a valid location to fire at.

Task 2.

For Strategy 1, the design paradigm for the **hunting mode** is **decrease and conquer** because the player is able to decrease the number of potential cells to check by firing only every second square.

Also **Targeting mode** is **brute force paradigm**, as it simply needs to check the four adjacent cells to the hit coordinate.

***Algorithm Strategy I-Hunt***

**Input** board[0,1,...,n−1][0,1,...,n−1], set of remaining ships, Sr ⊆ S, set of miss coordinates, M, set of hit coordinates, H.

**Output** fire the point of the board

**procedure** StrategyI\_Hunt(board,Sr,M,H)

**for** i ← 0 to n-1 by 2 **do**

**for** j ← 0 to n-1 by 2 **do**

**if** M.contains(i,j) == False and H.contains(i,j) == False **then**

return fire(board[j][i])

**end if**

**end for**

**end for**

**end procedure**

The time complexity of this algorithm is O(n2/4) → **O(n2)** in worst case, as it checks every second cell on the board.

***Algorithm Strategy I-Target***

**Input** board[0,1,...,n−1][0,1,...,n−1], set of remaining ships, Sr ⊆ S, set of miss coordinates, M, set of hit coordinates, H, the most recently hit coordinate, h.

**Output** fire the point of the board

**procedure** StrategyI\_Target(board,Sr,M,H,h)

targets ← [(h[0]-1, h[1]), (h[0]+1, h[1]), (h[0], h[1]-1), (h[0], h[1]+1)]

**for** target in targets **do**

**if** M.contains(target) == False and H.contains(target) == False **then**

return fire(board[target[1]][target[0]])

**end if**

return None

**end for**

**end procedure**

The time complexity of this algorithm is **O(1)** because it only checks the adjacent cells to the hit coordinate.

Task 3.

The underlying design paradigm of Strategy 2 is **divide and conquer**.

The algorithm divides the board into cells and calculated the probability of a ship occupying each cell, then uses this information to select the most likely cell to fire at in the hunting mode and most likely adjacent cell to fire at in the targeting mode.

By dividing the board into the cells and calculating the probability of ship placement for each cell, the algorithm is able to reduce the search space and make more informed decisions about where to fire.

***Algorithm StrategyII-Hunt***

**Input** board[0,1,...,n−1][0,1,...,n−1], set of remaining ships, Sr ⊆ S, set of miss coordinates, M, set of hit coordinates, H.

**Output** fire the point of the board

**procedure** StrategyII\_Hunt(board,Sr,M,H)

counts← calculateConfigurationCounts(board, Sr,M,H)

target ← select\_highest\_count (board,M,H,counts)

return fire(board[target[1]][target[0]])

**end procedure**

**procedure** select\_highest\_count (board,M,H,counts)

max\_count ← max(counts), max\_cells = []

**for** i←0 to n-1 **do**

**for** j←0 to n-1 **do**

**if** M.contains(i,j) == False and H.contains(i,j) == False and counts[i][j] == max\_count then

max\_cells.append((i,j))

**end if**

**end for**

**end for**

sort(max\_cells)

return max\_cells[0]

**end procedure**

**procedure** calculateConfigurationCounts(board, ships,M,H)

counts : Array[n][n]

counts.clear() // initialize as 0 for each value of this array

**for** ship in ships **do**

**for** i←0 to n-1 **do**

**for** j←0 to n-1 **do**

**for** orientation in ['horizontal', 'vertical'] **do**

**if** can\_fit\_ship(board, ship, i, j, orientation,M,H) **then**

**for** cell in get\_ship\_cells(ship, i, j, orientation) **do**

counts[cell[0]][cell[1]] += 1

**end for**

**end if**

**end for**

**end for**

**end for**

**end for**

**return** counts

**end procedure**

**procedure** can\_fit\_ship(board, ship, i, j, orientation,M,H)

**if** orientation == 'horizontal' **then**

**if** j + f(ship)[0] > n **then**

**return** False

**for r**←0 **to** f(ship)[0] **do**

**for c**←0 **to** f(ship)[1] **do**

**if** board[i+r][j+c] is not None **then**

**return** False

**if** M.contains(i+r,j+c) == False **then**

**return** False

**return** True

**elif orientation == 'vertical':**

**if** i + f(ship)[1] > n **then**

**return** False

**for r** ←0 **to** f(ship)[1] **do**

**for c** ←0 **to** f(ship)[0] **do**

**if** board[i+r][j+c] is not None **then**

**return** False

**if** M.contains(i+r,j+c) == False **then**

**return** False

**return** True

**end procedure**

**procedure** get\_ship\_cells(ship, i, j, orientation)

cells = []

**if** orientation == 'horizontal' **then**

**for r** ←0 **to** f(ship)[1] **do**

**for c** ←0 **to** f(ship)[0] **do**

cells.append((i+r,j+c))

**else**

**for r** ←0 **to** f(ship)[0] **do**

**for c** ←0 **to** f(ship)[1] **do**

cells.append((i+r,j+c))

**end if**

**end procedure**

Here f(ship)[0] means its width and f(ship)[1] means its height.

In above algorithm, calculateConfigurationCounts function calculates the number of ship configurations that can occupy each cell on the board and stores this information in a 2D array called “counts”.

It then finds the cells with the highest count and selects top left point of these cells to fire at using select\_highest\_count function.

So the time complexity depends on calculateConfigurationCounts function because this time complexity is bigger than the rest functions’s time complexity O(n2).

The calculateConfigurationCounts function takes a board and a list of ships as input, and returns a 2D array representing the count of possible ship configurations for each cell on the board.

The can\_fit\_ship function takes a board, a ship, a row index, a column index, and an orientation as input, and returns a boolean indicating whether the ship can be placed on the given cell in the given orientation.

The time complexity of calculateConfigurationCounts function is O(n4), where n is the length of the board. This is because the function iterates over all cells on the board, all ships, and all possible orientations. However, optimizations can be made to reduce the number of iterations, such as only iterating over cells that have not yet been fired upon.

In get\_ship\_cells , we compute the coordinates of all cells of the ship based on its shape and orientation.

***Algorithm StrategyII-Target***

**Input** board[0,1,...,n−1][0,1,...,n−1], set of remaining ships, Sr ⊆ S, set of miss coordinates, M, set of hit coordinates, H, the most recently hit coordinate, h.

**Output** fire the point of the board

**procedure** StrategyII\_Hunt(board,Sr,M,H.h)

targets ← [(h[0]-1, h[1]), (h[0]+1, h[1]), (h[0], h[1]-1), (h[0], h[1]+1)]

target\_counts ← []

H.append(h)

counts ←calculateConfigurationCounts(board, ships,M,H)

**for** target in targets **do**

**if** M.contains(target) == False and H.contains(target) == False **then**

target\_counts = counts(target[0], target(1))

**else then**

target\_counts.append(0)

**end if**

**end for**

index = max(target\_counts).index()

x = targets[index][0]

y = targets[index][1]

**return** fire(board[y][x])

**end procedure**

The complexity of this algorithm is O(n^2), where n is the size of the board, since it iterates over each adjacent cell to the hit cell. However, the actual time complexity may be less since not all adjacent cells need to be considered based on the shape of the hit ship.

If the ships can have any arbitrary shape, the time complexity of calculateConfigurationCounts would depend on the algorithm used to check if a ship can be placed on a cell, but it would still likely be at least O(n4) so the entire time complexity will be at least O(n4).

Task 4.

**Case 3:**

1. Using above algorithm, we can calculate the probability distribution of the remaining cells in the board.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 2 | 3 | 2 | 3 | 3 |
| 2 | 3 | O | 3 | 5 | 5 |
| 4 | 3 | X | 4 | 8 | 6 |
| 5 | 5 | X | 6 | 11 | 6 |
| 5 | 5 | X | 6 | 10 | 6 |
| 3 | 3 | X | 3 | 6 | 3 |

1. So the current highest probability in this board is (2,4).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 2 | 3 | 2 | 3 | 3 |
| 2 | 3 | O | 3 | 5 | 5 |
| 4 | 3 | X | 1 | 4 | 1 |
| 5 | 5 | X | 3 | O | 3 |
| 5 | 5 | X | 1 | 5 | 1 |
| 3 | 3 | X | 3 | 6 | 3 |

**Case 5:**

1. **3)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 5 | 5 | 4 | 2 | 2 |
| 3 | 5 | 5 | 5 | 3 | 2 |
| X | X | X | X | O | 3 |
| 3 | 6 | 6 | 4 | 3 | 2 |
| 6 | 9 | 9 | 8 | 4 | 4 |
| 3 | 6 | 6 | 6 | 5 | 3 |

1. The current highest probability in this board is (1,1).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 5 | 5 | 4 | 2 | 2 |
| 3 | 5 | 5 | 5 | 3 | 2 |
| X | X | X | X | O | 3 |
| 1 | 4 | 1 | 4 | 3 | 2 |
| 4 | O | 5 | 8 | 4 | 4 |
| 1 | 4 | 1 | 6 | 5 | 3 |

Task 5

**Algorithm SunkShips:**

**Input:** board[0,1,...,n−1][0,1,...,n−1], set of ships, S, set of miss coordinates, M, set of hit coordinates, H.

**Output:** The number of the ships which are sunk and names

**procedure** SunkShips (board,S,M,H)

sunkships ← [] , markedHitPoints ← [], sunkNum ← 0

**for** h in H **do**

**if markedHitPoints.contains(h) == False**

linkedCells ←[]

DFS(board,H,markedHitPoints,h,linkedCells)

type ← getTypeOfCells(linkedCells,S)

**if** type is not None **then**

S.remove(type)

sunkships.append(type)

sunkNum ← sunkNum + 1

**end if**

**end for**

**return** sunkNum, sunkships

**end procedure**

**procedure DFS(**board,H,markedHitPoints,h,linkedCells**)**

linkedCells.append(h), markedHitPoints.append(h)

directions ← [[-1,0]. [0,1], [1,0], [0,-1]]

**for** dir in directions **do**

tempPoint ← (h[0] + dir[0], h[1] + dir[1])

**if** H.contains(tempPoint) and markedHitPoints.contains(tempPoint) == False and tempPoint[0] >= 0 and tempPoint[0] < n and tempPoint[1] >=0 and tempPoin[1] <n **then**

DFS(board,H,markedHitPoints,tempPoint,linkedCells)

**end if**

**end for**

**end procedure**

**procedure** getTypeOfCells(linkedCells,S)

shipInfo← []

**for** ship in S **do**

shipInfo.append({type : ship.type, size; ship.size, width: f(ship)[0], height : f(ship)[1]})

**end for**

**for** ship in shipInfo **do**

**if** linkedCell.size == ship.size **then**

**if** linkedCell.width == f(ship)[0] and linkedCell.height == f(ship)[1] **then**

**return** S.type

**if** linkedCell.width == f(ship)[1] and linkedCell.height == f(ship)[0] **then**

**return** S.type

**end if**

**end for**

**end procedure**

The data structure used in this algorithm is a set of dictionaries to keep track of ship types,their cell counts,width and height. (See getTypeOfCells() function)

Also this algorithm uses DFS and its time Complexity is O(mp) here m is the average size of a ship in S and p is the number of hit coordinates in H.

The space complexity is also proportional to the number of hit cells (p).

Task 6

1. This algorithm is almost correct and it’s similar with my above pseudo code.

Here the flood fill search is similar with DFS, and I used getTypeOfCells() function to identify the size and shape of the ship.

1. The time complexity of the algorithm is O(n^2), where n is the maximum of the number of rows and the number of columns of the board. The algorithm visits each cell at most once and performs a constant amount of work for each cell. The space complexity is O(n^2) due to the visited set and the ship\_cells list.

However, this algorithm may not work correctly in all edge cases.

If the part of the larger ship is hit , then we can mistakenly identify the sunkship.

For.e.g if 2 cells of submarine are hit, we can be confused the KoKo is sunk.

**procedure** FloodFill (board,S,M,H,linkedCells, markedHitPoints)

linkedCells.append(h), markedHitPoints.append(h)

directions ← [[-1,0]. [0,1], [1,0], [0,-1]]

**for** dir in directions **do**

tempPoint ← (h[0] + dir[0], h[1] + dir[1])

**if** H.contains(tempPoint) and markedHitPoints.contains(tempPoint) == False and tempPoint[0] >= 0 and tempPoint[0] < n and tempPoint[1] >=0 and tempPoin[1] <n **then**

FloodFill (board,H,markedHitPoints,tempPoint,linkedCells)

**end if**

**end for**

**end procedure**

Task 7

We can find sunkships count using below algorithms.

**Algorithm SunkShips:**

**Input:** board[0,1,...,n−1][0,1,...,n−1], set of miss coordinates, M, set of hit coordinates, H.

**Output:** The number of the ships which are sunk

**procedure** GetShipsCount (board,M,H)

markedHitPoints ← [], sunkNum ← 0

**for** h in H **do**

**if markedHitPoints.contains(h) == False**

linkedCells ←[]

DFS(board,H,markedHitPoints,h,linkedCells)

If(linkedCells.length > 1) sunkNum ← sunkNum + 1

**end for**

**return** sunkNum

**end procedure**

**procedure DFS(**board,H,markedHitPoints,h,linkedCells**)**

linkedCells.append(h), markedHitPoints.append(h)

directions ← [[-1,0]. [0,1], [1,0], [0,-1]]

**for** dir in directions **do**

tempPoint ← (h[0] + dir[0], h[1] + dir[1])

**if** markedHitPoints.length > 19 **then**

**continue**

**if** H.contains(tempPoint) and markedHitPoints.contains(tempPoint) == False and tempPoint[0] >= 0 and tempPoint[0] < n and tempPoint[1] >=0 and tempPoin[1] <n **then**

DFS(board,H,markedHitPoints,tempPoint,linkedCells)

**end if**

**end for**

**end procedure**

Time complexity: O(n2) where n is the length of the board. This is because the algorithm goes through every cell in the board up to n2 times, depending on the number of hits and misses.