Queen's UNIVERSITY

GRP 21: Battleship - Best Hit Detector

Donovan Bisson (19dcb2)

Matthew Woo (20mw6)

Justin Woo (20jw8)

Marc Bucovy (19meb12)

Course Modelling Project

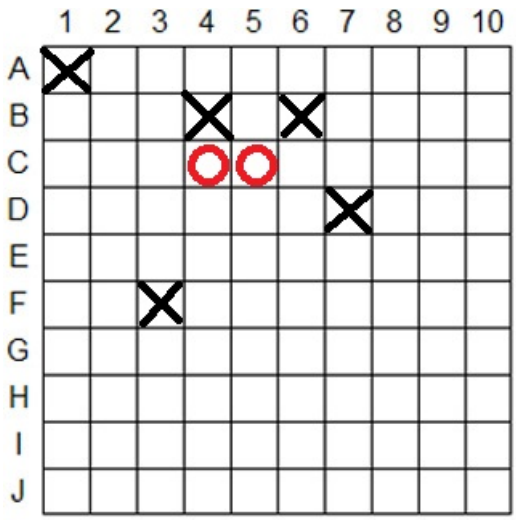
CISC/CMPE 204

Logic for Computing Science

December 14, 2021

Abstract

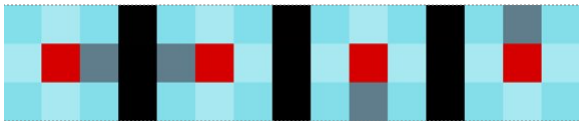
Battleship is a two-player board game about sinking all the opposing player's ships by taking turns with them attacking certain grid coordinates on each other's board. The positions of each player's ships are unknown to the opposing player. The picture below depicts the grid coordinates that one of the players have attacked. The black X marks attacks that did not hit any ship; a miss. Red circles indicate an attack has hit a ship. Blank spaces are areas of the grid that are yet to have been attacked. Ships can be adjacent to one another, but cannot overlap or go outside the bounds of the board.



An example of a typical battleship game (In our model we will only be using number coordinates)

This project intends to show the likelihood of a ship being hit by attacking a specified tile on an arbitrary board state. An example of an arbitrary board state would be the grid displayed above. The standard Battleship pieces will be used (2 x 1 Destroyer, 3 x 1 Submarine, 3 x 1 Cruiser, 4 x 1 Battleship, 5 x 1 Carrier), as well as other standard Battleship rules (10 x 10 grid, no overlapping ships, no diagonals). The only exception to this will be that ships will not be named on hit; however, they will still be named when sunk.

This is done by examining every possible permutation for every ship that have a part of the ship presiding on the checked tile. Every tile is checked this way, and the tile with the most permutations has the greatest chance of a ship presiding on that checked tile. These permutations are affected by information on the board such as misses and hits at certain grid coordinates. Thus a tile or tiles will be considered the best places to attack to increase the likelihood of a hit. The coordinates will be in numbers with the top left tile being (0,0) and the bottom right tile being (9,9)



The diagram above depicts how the permutations will be checked at a single tile. By following the battleship rules for the ship lengths stated above, there would be a total of 4 + 6 + 6 + 8 + 10 = 34 permutations that a single tile can be checked for. By analyzing which tiles can cover the most of these permutations, it determines the best tiles to hit.

Propositions

D, S, Cr, B, Ca - Each evaluates to true when the respective vessel (Destroyer, Submarine, Cruiser, Battleship, Carrier) that the proposition represents is active on the board. By being marked as false, the permutations of the ship will not be used during the hit check process.

$D_{i,j,d}, S_{i,j,s}, Cr_{i,j,cr}, B_{i,j,b}, Ca_{i,j,ca}$ - These are the permutations of each ship type relative to (i,j). The third value represents the possible orientations of the ship.

$X_{i,j}$ Evaluates to true when the point (i,j) is a known, unresolved hit. Unresolved means a hit on a ship but the type of the ship is still not known.

$O_{i,j}$ Evaluates to true when the point (i,j) is known, but is not an unresolved hit (Miss, Out of Bounds, confirmed sink). This allows the identification of ship permutations that are not possible based on these tiles where something other than an active ship part is located

Constraints

Dimensions of the board are defined as $\{(i,j)|1 \geq i \geq m, 1 \geq j \geq n\}$ where m and n are the width of the board.

$X_{i,j} \rightarrow P_D \vee P_S \vee P_{Cr} \vee P_B \vee P_{Ca}$.
P is shorthand for all permutations. ie. $P_D = (D(i,j,1) \vee D(i,j,2) \vee D(i,j,3) \vee D(i,j,4))$. Unresolved hit must allow a permutation based on any of the 5 ships at the coordinate of (i,j).

For any $D_{i,j,1}, D_{i,j,1} \rightarrow D \wedge X_{i,j} \wedge X_{i+1,j}$

For any (i,j), $\neg(X_{i,j} \wedge O_{i,j})$. Means that a tile cannot be both a unresolved hit and Not an unresolved hit

Model Exploration

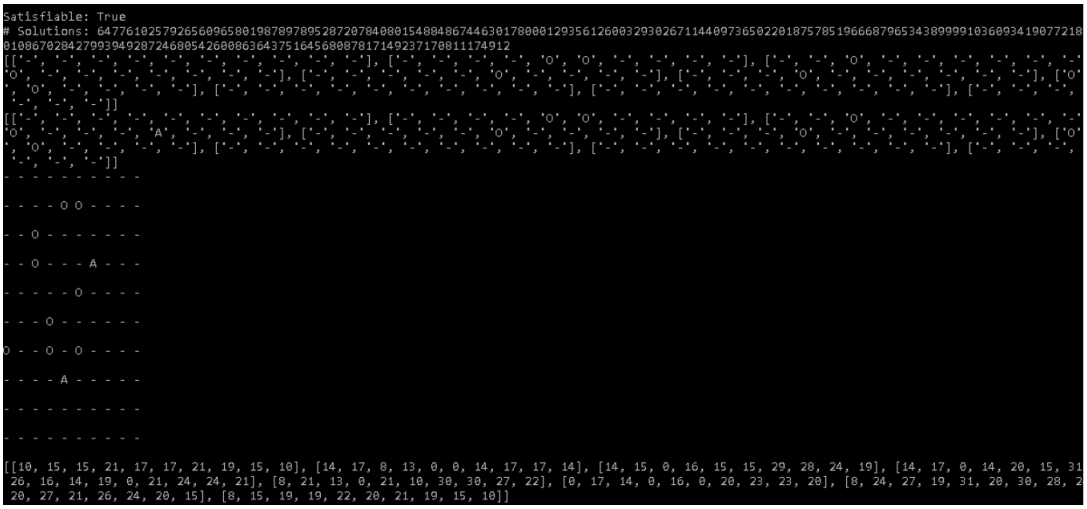
We first intended to have separate classes to represent the hits and misses. To simplify this we made the Hit and Miss propositions a subclass of the coordinate class, which holds the coordinate location on the board.

We also wanted to have one class to model all permutations over all ship types. Instead of this, there is a distinct class for each ship type, within which each permutation for this specific type is to be held. In order to not introduce too much complexity, each ship type is a subclass of the ship class.

For each ship we created a function to deal with each hit, or assumed hit. These functions should rule out permutations, in relation to the coordinate being considered. By creating a function that works on each coordinate, we can further break down into a series of cases. Relative to the coordinate we search for instances where a permutation results in both a Hit and Miss proposition overlapping.

To further simplify the construction of the constraints, we decided to put these functions specific to the ships, into the classes specific to the ships. This way we will not have to repeatedly call the functions for each point, and the constraints should be made at the same time as the propositions. In this way we can rely on the coordinate class to do a lot of the work for us.

As we further worked on the code we found that the original permutations for the ships we set up was incorrectly done since we did not fully understand Bauhaus yet at the time so we had to rework them. Furthermore, we attempted to get our program to return truth values based on whether the tiles were set as misses.



Prototype version of our output. Note that the A tiles(suggested attack) are actually in incorrect spots. A consequence of obtaining x and y values in python incorrectly.

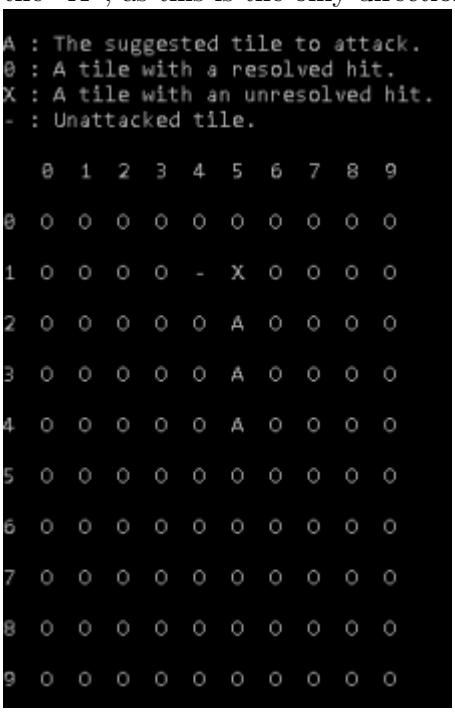
We also found that our constraint that states a hit and miss proposition can't both be true on the same tile is effectively redundant since when you initialize the board, it is impossible to have a tile at a coordinate be several different types at once. Though we kept this just to prevent some sort of random bug where a tile has both a hit and miss at the same time from occurring.

The run time of the program varies greatly, from under a minute to over 5 minutes. The time is very dependent on how many ships are active on the board, and what they are. This is because for the active ships, all permutations of each ship is created, for each valid coordinate across the board. To then test the truth of each of these permutations, for the relevant coordinates to each permutation, the truth values of "X" and "O" are compiled into constraints. In this way there is a constraint for every permutation, for every active ship, for every valid coordinate. The number of constraints is therefore greatly reduced when there are fewer active ships.

Before the outputting a solution, the user must input the board state they wish to be considered, and whether each ship is sunk or still active on the board. The final output marks the spaces with the highest probability of containing a hit.

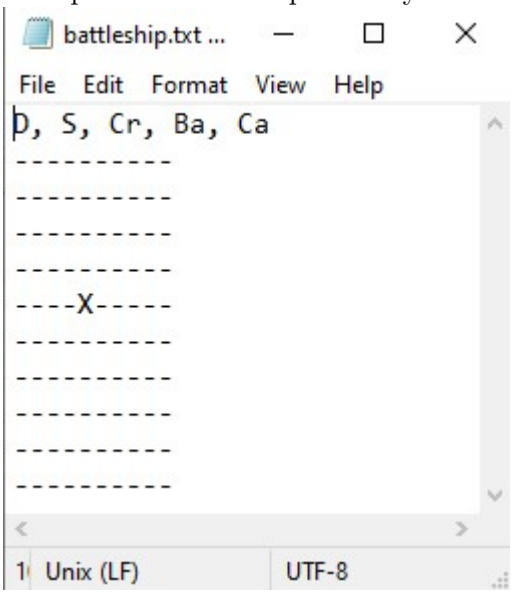
In the below case, the destroyer is the only active ship, and there is only one coordinate that is possible. This coordinate is therefore marked "A".

In the next case, the Battleship is the only active ship, and there is one space with an active hit. Therefore the only possible coordinates are below the "X", as this is the only direction the Battleship will fit.



We found that our original method of obtaining user input was too tedious for the user to use since it would force them type in each individual value one at a time every time they wanted to run the program so we opted for a text file based input. This will allow user input too be sent much faster as well as simpler for making small adjustments to a board that a user may want to change. You more easily read and fix mistakes of your board where as when you had to type it in it was much harder to see if you made mistake. It's an improvement to quality of life for the user.

Below is an example of how the input is set up, the first row contains the ships that are still active. The board below is pretty much the same as the example board shown previously.



Jape Proofs

A{1,2,3,4} represents a miss tile above,left,right and below a given tile
A represents a miss tile on the given tile
B{1,2,3,4} represents a hit tile above,left,right and below a given tile
B represents a hit tile on the given tile
D{1,2,3,4} represent destroyer permutations up,left,right,down on a given tile
D represents the destroyer ship proposition

This proves that if there is a miss tile below a given tile and no other tile is miss tile and the destroyer is still active then the only possible destroyer permutations are up,left and right.

1:	¬A, ¬A1, ¬A2, ¬A3, A4, D	premises
2:	(¬A1∧¬A∧D)→D1, (¬A2∧¬A∧D)→D2	premises
3:	(¬A3∧¬A∧D)→D3, (A4∧¬A∧D)→¬D4	premises
4:	A4∧¬A	∧ intro 1.5,1.1
5:	A4∧¬A∧D	∧ intro 4,1.6
6:	¬D4	→ elim 3.2,5
7:	¬A3∧¬A	∧ intro 1.4,1.1
8:	¬A3∧¬A∧D	∧ intro 7,1.6
9:	D3	→ elim 3.1,8
10:	¬A2∧¬A	∧ intro 1.3,1.1
11:	¬A2∧¬A∧D	∧ intro 10,1.6
12:	D2	→ elim 2.2,11
13:	¬A1∧¬A	∧ intro 1.2,1.1
14:	¬A1∧¬A∧D	∧ intro 13,1.6
15:	D1	→ elim 2.1,14
16:	D1∧D2	∧ intro 15,12
17:	D1∧D2∧D3	∧ intro 16,9
18:	D1∧D2∧D3∧¬D4	∧ intro 17,6

This proof represents the case where hit_detection is false. It shows that if the corresponding coordinates to the ships permutation are false for being a miss tile, then the permutation in question is true. In this case the destroyer is used as an example, so there are two coordinates taken into account.

1:	¬A, ¬A1, D, D→((¬A∧¬A1)→D1)	premises
2:	¬A∧¬A1	∧ intro 1.1,1.2
3:	(¬A∧¬A1)→D1	→ elim 1.4,1.3
4:	D1	→ elim 3,2

This proof represents the case where hit_detection is true. It shows that if the corresponding coordinates to a ships permutation are false for being a miss tile and one, or both, of the coordinates are true for being a hit, then the permutation is true. In this case the destroyer is used as an example, so there are two coordinates taken into account.

1: ¬A, ¬A1, ¬B, B1, D, D→(((¬A∧¬A1)∧(B∨B1))→D1) premises
2: B∨B1 ∨ intro 1.4
3: ¬A∧¬A1 ∧ intro 1.1,1.2
4: ¬A∧¬A1∧(B∨B1) ∧ intro 3,2
5: ((¬A∧¬A1)∧(B∨B1))→D1 → elim 1.6,1.5
6: D1 → elim 5,4

First-Order Extension

We could have quantifiers representing the ships that are active, as well as representing the tiles on the board.

Proposition(Predicate):
D(x), S(x), Cr(x), B(x), Ca(x) means x is a destroyer, submarine, cruiser, battleship, carrier. If no x satisfy the proposition, we can assume that this ship is no longer active.
X(i,j), i and j is a coordinate for a tile on the board with an unresolved hit
O(i,j), i and j is a coordinate for a tile that is out of bounds, or a miss, part of a confirmed sink.

Constraint:
$D_i = \{1, 2...m\}, D_j = \{1, 2...n\}$ where m and n are arbitrary integers
$\forall i \in D_i, \forall j \in D_j$, i and j act as coordinates for the board and must be a value in D_i and D_j

$\forall i.\forall j.((X(i,j) \rightarrow (\neg O(i,j)))$ For all i and j, if the tile at i,j is an known (resolved) tile. then the tile can't be a hit unknown ship.
$\forall i.\forall j.((O(i,j) \rightarrow \neg X(i,j)))$ For all i and j, if the tile at i,j is a hit on an unknown ship then the tile can't be an known (resolved) tile.
These 2 constraints would remove the need to check every single tile one by one to make sure that they're of a single type.

$\exists i, \exists j, \exists d((O(i,j-1) \vee O(i,j+1) \vee O(i+1,j) \vee O(i-1,j)) \rightarrow \neg D(i,j,d))$
Where i and j are coordinates of a tile and d is a permutation type of a destroyer (Using destroyer as an example). The constraint denotes that for any tile that borders an out of bound tile, a ship permutation will be false in the direction of said out of bound tile. This is useful for managing constraints on the vast amount of edge tiles. This can work for any ship because it focuses only on tiles one away but for further tiles it would require more copy paste predicates so we're just gonna leave this as is.