Author: Maxim Lapitan
Matriculation number: 22200839

Individual Paper (Battleship Puzzle)

1. Description of the problem

First of all, the problem that is taken as decision problem is **Battleship (puzzle version)**. In the following analysis it is going to be searched for a way to solve problems within different approaches. Such approaches as *Search Algorithm*, *Reinforcement Learning* and *SAT/SMT*. But before proceeding to the analysis of solutions to the problem in various ways, it is necessary to explain the **meaning** and **rules** of the problem.

The logical Battleship puzzle takes place on a 10x10 (or any other scale) grid-based playing field where 10 ships of different sizes are hidden. Versions of the game are considered classic and will stay the same within the whole analysis. So, ships are:

- 1 ship **4 squares** long;
- 2 ships **3 squares** long;
- 3 ships **2 squares** long;
- 4 ships **1 square** long;

Along the grid there are numerical indicators that illustrate the number of squares occupied by ship segments in a specific row or column. It would also be that water squares appear while solving the puzzle, where no ships were found.

Note that before game is started provided ships should be positioned on the puzzle-grid, meeting the following 4 conditions:

1. all ships are placed in a grid;
2. the dimensions of the original grid are kept;
3. no two ships occupy neighboring (orthogonal or diagonal) squares;
4. the number of ship segments in column (row) i is equal to the i-th value of the count column (row).

Basically, the ship on the final map is `'S'`. So, fundamental approach to solve Battleship puzzle involves replacing squares for unfinished ships (mark square with SHIP - `'S'`), replacing squares where water is predicted to be (mark squares with WATER - `'W'`), regarding third condition - replacing squares with water nearby fully destroyed ship if it is obvious that the ship was totally destroyed (replacing squares with WATER `'W'` nearby `'S'`) and searching for ships in a row or column where the number that corresponds to the number of squares containing ship segments. **This analysis will use a fundamental solution approach.**

2. Search as a Universal Problem-Solving Mechanism

To begin, the **Battleship puzzle** is constructively represented as a problem that can be solved by graph search. The cause for this is that there is an initial state, each state of the game formulates an actual layout of ships on the grid which takes us to a next state, and the target is to search for a state in which all ships are accurately identified without empty squares of water being identified as ships, and vice versa.

In general, graph search is possible here, because it helps to avoid visiting a state over and over again, keeping a record of previous states and only exploring the new ones. More important thing here is the edges between nodes. In this situation they represent a valid move that corresponds to discovering (revealing) a square on the grid.

Possible representation of final state of Battleship grid in python is:

```
final_state = [
    ['W', 'W', 'W', 'W', 'S', 'S', 'S', 'W', 'W', 'W',    3],
    ['S', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'S', 'W',    2],
    ['W', 'W', 'S', 'S', 'W', 'W', 'W', 'W', 'W', 'W',    2],
    ['W', 'W', 'W', 'W', 'W', 'S', 'S', 'S', 'S', 'W',    4],
    ['W', 'S', 'W', 'S', 'W', 'W', 'W', 'W', 'W', 'W',    2],
    ['W', 'S', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W',    1],
    ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'S', 'W',    1],
    ['W', 'W', 'W', 'W', 'W', 'W', 'S', 'W', 'S', 'W',    2],
    ['W', 'W', 'W', 'S', 'S', 'W', 'W', 'W', 'S', 'W',    3],
    ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W',    0],


    [ 1,   2,   1,   3,   2,   2,   3,   1,   5,   0        ]
]
```

initial_state

```
= [
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',    3],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',    2],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',    2],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',    4],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',    2],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',    1],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',    1],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',    2],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',    3],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',    0],


    [ 1,   2,   1,   3,   2,   2,   3,   1,   5,   0        ]
]
```

One possible state (node) is:

```
state = [
    ['W', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',    3],
    [' ', 'W', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',    2],
    ['W', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',    2],
    [' ', ' ', ' ', ' ', ' ', 'S', ' ', ' ', ' ', ' ',    4],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',    2],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',    1],
    [' ', ' ', ' ', ' ', ' ', 'W', 'W', 'W', ' ', ' ',    1],
    [' ', ' ', ' ', ' ', ' ', 'W', 'S', 'W', ' ', ' ',    2],
    [' ', ' ', ' ', ' ', ' ', 'W', 'W', 'W', ' ', ' ',    3],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',    0],
    [ 1,   2,   1,   3,   2,   2,   3,   1,   5,   0       ]
]
```

- one ship is identified
- one segment of ship is identified
- water is identified in several squares

Looking for a possible solution for this problem, it is clearly seen that ship segments can be placed absolutely in different ways. But not all states will be satisfiable. So, before describing possible functions that correspond to possible successors, valid moves and actual moves, it is better to explain what is needed from the search algorithm. The idea is not to find the shortest possible or optimal solution, but the deepest and longest one, that includes the appearance of all ships with given length, all numerical indicators satisfying the number of ship segments and all empty squares replaced with water. Most appropriate approach is **Depth-first** search that is implemented via **A star** by using a heuristic that prefers longer paths. Because it is better to try to explore the graph in a different order, extending a solution as far as possible and only trying another one if the current extension fails.

To solve such a problem the only thing needed is to spell out the graph involved, using only three functions (isGoal, nextStates and isValid).

A generic formulation of graph search requires:

- a way to control whether the goal has been reached:

isGoal: State -> Bool

> state should be equal to the final state (that is already given).

- a function that returns the possible successors of the current state:

nextStates : State → List(State)

> going through the grid, from left upper corner to left bottom corner, 1 by 1 square, replacing empty squares with `'W'` or `'S'` or couple squares with `'W'` by applying the function `isValid` that checks this very square for row and column numerical indicators.

- a function that returns either `'W'` or `'S'` or dictionary of coordinates, which should be replaced with `'W'` for the current square:

$$\text{isValid} : \text{State}, i, j \rightarrow \text{List(State)}, \text{Tuple}(i, j)$$

> function gets coordinates of a square, then gets very specific numerical indicators, analyzes if it is appropriate to place `water` or `ship` by applying two other functions. One of the functions is analyzing the whole state for the number of ships and its sizes, so that no 2 4-long ships appear. Second function analyzes if it is just one segment of a ship, rather it is the whole ship, then returns if it is possible to place `water` nearby a square with the ship. When the whole procedure is done `isValid` returns `'W'` or `'S'` or `'W'` for all nearby squares if the ship is fully found.

3. Reinforcement Learning

Now it is necessary to discuss the possibility of implementing the **Battleship puzzle** into Reinforcement Learning. First of all, **Battleship puzzle** is a kind of the game, so it is better to see how the MENACE approach can be involved in solving this very problem. The MENACE algorithm is just about selection of stochastic moves, credit assignment (reward or trial as winning moves, punishment or error as losing moves), and function approximation to facilitate efficient Reinforcement Learning behavior. All of these criterias make it a suitable candidate for solving complex problems such as the **Battleship puzzle**. But how is it going to be done?

Building MENACE

1. Policies

In MENACE it is always about the matchbox system. So it is necessary to define its matchbox input and output. The matchbox system tells us what action to take in a given state. Such a function is called a **policy**. Policy for this very problem is deciding which square to reveal in order to identify all ships on the grid.

> Battleship puzzle (Policy): State = Field (or simply Grid), Action = ChooseSquare

This policy can be considered as sufficient one, because it attains both a long-term goal and an immediate one as well. **Long-term goal** in Battleship puzzle: minimize the number of revealed squares by generating a strategy of which squares should be opened, so that the game is ended in a minimized number of actions. **Short-term goal** is: reveal big ships, so that the biggest possible number of squares are revealed in the shortest possible number of actions.

Such goals can be easily explained. For example, in the search algorithm it is better to consider the number of ships' segments, because we better find all possible states and find one that is the same as initial state, but while talking about reinforcement learning, we can search for an absolute strategy that can be applied for every possible state in this game. The reason for such an assumption is its ability to learn while randomly revealing squares.

2. Single Matchbox

For each conceivable move, the current valuation of the system is represented by a single matchbox. Better moves are correlated with larger values. A value function is implemented by the matchboxes. Battleship' equivalent of matchboxes: a function that determines the worth of revealing squares on the grid, based on 4 conditions.

State-action value function in Battleship is:

> value : (Field, ChooseSquare) -> Number

The grid states can then be used to index a Python dictionary that represents the MENACE system:

```
menace = {}
```

A string-list pair can be used to represent a battleship matchbox:

- The string is the state of the grid;

```
state = "".join(["_" for _ in range(100)])
```

- The list has as many elements as the string has unrevealed "squares".

```
weights = [100 for _ in state if _ == "_"] # 100 by default
menace[state] = weights
'Every element is a natural number, signifying the significance of a sufficiently
revealed square connected to the corresponding "square" that has not yet been
revealed.'
```

3. Function Approximation

Function approximation in Battleship is implemented via trial and error by loading a thousand of different possible ships placement and running algorithm more than 10 thousands times for each grid.

New function `NewGrid` is initialized in order to create new grid. It does not need any pre-requirements. It just creates a new 10x10 grid with 10 ships of different sizes. This function places ships on the grid in a random order every time the function is run, but all the necessary conditions for placing ships are preserved.

4. 1. Choosing a Move

Chosen move is made chaotically by applying the function `ChooseSquare`. Moves with higher values — that is, more matchbox representatives — have a higher probability of getting selected.

> ChooseSquare : Grid -> RevealSquare

On a map, this method chooses a random square. Because it only selects between empty squares, it is unable to choose squares that have already been shown.

4. 2. Credit Assignment

Once every ship has been identified, any open squares without any ships are penalized. Giving credit or assigning blame to a particular exposed square is not always easy. Well exposed squares get good results, though not always. Stochastic selection reduces the likelihood of eliminating strongly exposed squares.

4. 3. Rules Based Functions

First of all, due to the way the ships are positioned, no square close by can be seen if the ship is completely destroyed. The function receives the grid and continuously determines if the ship was completely destroyed. If so, the weights that are unique to squares are removed from the list of weights. Thus, adjacent cells are hidden.

> DeadShip: Grid -> DeleteWeights

Nextly, only surrounding cells should be shown if the ship's section is revealed in order to determine whether any more ship segments remain.

> ChooseNear: Grid -> ChooseSquare: Tuple(coordinates)

Finally, a list is made that keeps track of the ships that are still on this grid. When a ship of size 1 is destroyed, for instance, {ships[0] -= 1} and so forth. With this feature, it would be more evident for MENACE to stop displaying the squares in the event that all of the ships had been located.

```
ships = [4,3,2,1] # where 4 = 1-size, 3 = 2-size, 2 = 3-size, 1 = 4-size
```

Additionally, a function that updates ships and obtains the grid while the game is running. The updated ships array is returned.

> CheckShips: Grid -> ships

5. Update Rule

After every game, this function will update the weights (the game ends when all ships have been found). Along with history with state and disclosed square, it receives a menace dictionary with all states and weights. Subsequently, weights in squares where ships were concealed are upgraded. The squares' weights that were discovered close to ships have also been improved, though not as much as in previous statement. The squares' weights that were discovered close to ships are punished. Square weights that were discovered too far away from ships are penalized significantly more severely than in previous statement.

> UpdateMenace: (menace, history, ships_location) -> menace

4. SAT and SMT

The primary issue with SAT/SMT is whether or not it can be resolved. It is necessary to ascertain whether the Battleship puzzle problem is NP-complete in order to determine with certainty whether it can be solved (in theory) using SAT/SMT. The saying "SAT is the first problem that was proven to be NP-complete" [1] explains why these claims are made. In addition, it is claimed that "If we could find solutions of some NP-complete problem quickly, we could quickly find the solutions of every other problem to which a given solution can be easily verified". [2]

This implies that if Battleship is an NP-complete problem, it can be quickly and readily confirmed for other NP-complete problems (namely, SAT/SMT). Is the Battleship puzzle, however, NP-complete? Yes, the prove is already provided, see document. [3] Nevertheless, it is still necessary to prove the possibility of implementation.

First of all, the SAT/SMT technique functions in this instance by converting the rules of the Battleship puzzle into a set of logical constraints. Thus, all prerequisites must be indicated prior to the primary rules being implemented.

Boolean variables, which are integrated z3 variables, can be used to represent the initial state (grid), which is just a field or map. Every grid cell has an associated Boolean variable that indicates whether the cell contains a ship segment ({`'S'`} - True) or water ({`'W'`} - False).

Secondly, let's talk about set of constraints. It would be checked whether Battleship puzzle satisfies a set of properties (namely rules). There are only few expected **constraints**:

1. Ship Size Constraint

This very constraint checks whether all required ships are positioned on the grid. In Battleship puzzle this number is 10, where it should accounts for 1 {*4-segments*} ship, 2 {*3-segments*} ships, 3 {*2-segments*} ships and 4 {*1-segment*} ships. This constraint does not count overall number of segments on the grid, but follows the grid both horizontally and vertically to ensure that all required ships with their prescribed sizes are placed correctly. What does it include - to be placed correctly? It denotes - `'S'` placed near each other, without any overlap or gaps.

2. Ship Count Constraint

In the case of this constraint, it sum ups appearance of all true Boolean variables on the map, ensuring that number of ship segments is 20. For example, it can be represented in such way:

```
Sum([If(grid[i][j], 1, 0) for i in range(rows) for j in range(cols)]) == 20
```

That means - if the total sum of true variables on the map is 20, then it allows model to be satisfiable.

3. Initial State Constraint

This restriction examines the puzzle's initial state for given ship or water squares. In the event that any such previously disclosed squares exist on the map, they are converted into matching Boolean variables. It is required in cases like this when some of the squares have already been made known and should be considered as well while designing the model.

4. Ship Placement Constraint

Such constraint represents the inability of ships to be placed next to or overlap one another. It means that no two ships occupy neighboring (orthogonal or diagonal) squares. They also must be arranged either horizontally or vertically. To do this, another set of limitations that preserve these rules is needed, because only one is not enough.

Namely, it needs 3 constraints. First is required to check horizontally for ships' placement. Second one is needed to verify vertical adjacency of ships. And last one is necessary to delete cases of orthogonal or diagonal neighboring of two ships.

5. Column and Row Constraint

From the very bedinning of this analysis, it was previously said that numerical indicators are part of the puzzle. They play significant role while talking about SAT/SMT approach, because they directly indicate required number of ships' segments in rows and columns. So, this imposed constraint is allowed to ensure these counts are met while modeling the solution. It helps a lot, because it prevents chaotic placement of ships and makes each provided possible state unique.

Puzzle Solving Overview

Solution may be found only in a sitution when all previously described constraints are satisfiable. In a case when solution is found by solver, human-recognisable grid is translated back,and squares with identified water and ships are illustrated. However, sometimes solution may not exist. In this case, it is indicated that the Battleship puzzle with predefinied rules is unsolvable.

While talking about implementation, it is always easier to show possible integration of said above words in some code. So, it may look like this little **pseudocode**:

```
function battleshipSolver whith provided {grid}
    1. initialize solver
    2. create Boolean variables for each grid cell

    3. apply constraints:
        3.1 initial state translation
        3.2 total ship count
        3.3 ship size and placement
        3.4 adjacency rules
        3.5 row and column constraints based on initial state

    4. if solver finds a solution:
        translate the solution into the grid format
        return solved grid
    else:
        return no solution found
```

5. References

1. Wikimedia Foundation. (2023, December 20). Boolean satisfiability problem. Wikipedia. https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

2. Wikimedia Foundation. (2024, January 6). NP-completeness. Wikipedia. https://en.wikipedia.org/wiki/NP-completeness

3. Sevenster, M. 2004, 'Battleships as Decision Problem', ICGA Journal [Electronic], Vol. 27, No. 3, pp.142-149. ISSN 1389-6911.