Description of the problem:

We can describe the problem statement as follows: our agent is in charge of rescuing passengers on sinking ships. In order to achieve this, the priority of the agent is to pick up passengers from these boats and retrieve the black boxes that are available once the ship contains no passengers. When a ship sinks and has no more passengers, it turns into a wreck, however our agent is still responsible for getting its black box before it is damaged. Every time step, all ships lose a passenger, and if the ship is a wreck the blackbox gains one more damage. A time step is counted everytime our agent performs an action. We reach our goal state if there are no more passengers to rescue and no more undamaged black boxes to retrieve and our agent is not carrying any passengers. Our goal is to maximise the number of passengers saved and the number of the retrieved black boxes.

Data Types:

Search Tree Node ADT:

- State class which represents the possible nodes creation. Takes the cgI and cgJ of the coast guard which are its position in the grid, cgPass which is the capacity of the agent, int rescued and int retrieved representing the number of passengers. Then an arraylist of the Ships available in the grid, ArrayList of strings that keeps in track the list of actions performed by the agent until this state and lastly a string that denotes the action that the agent wants to perform from this node, which will represent a child state to this node.

Search Problem ADT:

- Ship class takes int i and j for the location of the ship and the number of passengers.
- Station class that has a constructor that takes into consideration the i and j for the position of the stations in the grid
- Cell class takes i and j to denote the position in the grid
- Grid takes m and n for the dimensions of the grid, ArrayList of ships and Arraylist of stations where the sizes of both ArrayLists are to be randomised upon creation of the grid.

Coast Guard Problem ADT:

The constructor of the coast guard class takes as parameters the i and j which indicates the position of the coast guard, the Grid and the maximum capacity of the coast guard.

Main Functions:

GenGrid:

We implemented a function that generates a grid randomly by assigning random variables for the m and n, randomising the number of ships and the number of stations and placing

them randomly in the grid then placing our agent in any empty cell. We ensure that no two elements are to be generated in the same cell. The method returns a string that gets passed later for the solve function.

Solve:

The solve function is the driver function of our implementation, it takes as an input a string describing the grid, a string representing the search algorithm that the method will run, and a boolean indicating whether or not we need to visualise our search or not.
The first thing our function does, is to split the grid string to generate from it the necessary data structures needed for our problem, such as initialising our grid with the appropriate size, initialising the position and the capacity of our coast guard, the list of ships with their corresponding locations and the passengers on board, the locations of the stations.
Then we create a state with these retrieved variables to generate our initial state that will be the root of the search tree in our problem.
Based on the strategy string, a search algorithm is being chosen to solve our search problem.

Search Algorithms:

Search algorithms have a similar implementation depending on the queueing function for each strategy. But, essentially, as discussed in the lecture, the general search creates a data structure for the nodes based on our queueing function to enqueue the root and search for other nodes that pass the goal test. The search goes on until our data structure does not have any nodes to remove or we found a node that satisfied the goal test. Each iteration, the search algorithms checks to see all possible actions that can be taken from this state, these represent the children of the current state. The children are to be considered for expansion only if they pass some efficiency tests such as; a non movement action can not occur twice (meaning that the coast guard agent can not for example perform the action drop twice). Similarly, actions that undo the movement of the agent are not to be performed, such as allowing the agent to move up and down repetitively without having an impact on its final position. These efficiency tests ensure that no unnecessary state will be expanded to allow for a better time efficiency.

Breadth First Search:

Taking a look specifically on the breadth first search algorithm, the chosen data structure for our queueing function is a regular queue from a linked list, we first enqueue the root using .add() then we start the while loop saying that we should stop if our queue is empty. Then we pass a check on every possible action the agent can perform to see if a node is eligible to be enqueued on the queue or it needs to be skipped. We have a stopping condition inside the loop that is our goal test, meaning the searching should stop if there are no more passengers alive, no more retrievable black boxes, the agent is not carrying any passengers to drop.
Then we return a string[] that has the expansion sequence, the number of retrieved boxes, the number of dead passengers and the number of expanded nodes.

Depth First Search:

Taking a look specifically at the depth first search algorithm, the chosen data structure for our queueing function is a stack, we first push the root using .push() then we start the while loop saying that we should stop if our queue is empty. Then we pass a check on every possible action the agent can perform to see if a node is eligible to be enqueued on the queue or it needs to be skipped. We have a stopping condition inside the loop that is our goal test, meaning the searching should stop if there are no more passengers alive, no more retrievable black boxes, the agent is not carrying any passengers to drop.
Then we return a string[] that has the expansion sequence, the number of retrieved boxes, the number of dead passengers and the number of expanded nodes

Iterative Deepening Search:

The iterative deepening search is basically the DFS but with restrictions on the depth of our search. Hence, we implemented a helper method which basically mimics the dfs by passing it the specific level that we want to stop the search at, then we handle the increments of the levels in the driver function which starts over the search if we reach the end of the depth with no goal state by calling the helper function with a higher order of depth.
Then we return a string[] that has the expansion sequence, the number of retrieved boxes, the number of dead passengers and the number of expanded nodes

Greedy Search and AStar:

We implemented 2 functions for each search which get called based on the chosen heuristic. The data structure used here is a priority queue that calles a compare function which decides which path to take based on the defined priorities depending on every heuristic in the compare method.
The difference between Greedy and AStar, is that AStar takes into consideration the path coast, where the greedy algorithm only takes into consideration the heuristic. We initialise both in the compare function and trigger the coast with a boolean variable if the chosen algorithm is AStar.
In both cases, we return a string[] that has the expansion sequence, the number of retrieved boxes, the number of dead passengers and the number of expanded nodes.

Performance comparison:

We can observe that, Iterative Deepening takes the longest CPU while the gready algorithms have the shortest CPU time.
In terms of Memory, BFS consumes the most RAM and DFS has the lowest memory rate.
Lastly, taking a look at the expanded nodes, BFS has the highest number of expanded nodes and DFS has the lowest number of expanded nodes.

Heuristics:

Heuristic 1:
Our first heuristic is taking into account with every time step the number of possible retrievable black boxes.
In the case of the AStar Algorithm, we add to this value the path coast which is equivalent to the depth of my path.

Heuristic 2:
In the 2nd heuristic, we take into account the number of the passengers left to save and we divide them with the capacity of our agent. In order to not reach an h(x)=0 in a state which is not the goal state, we add to this the number of black boxes left to retrieve.