- A brief discussion of the problem

Our problem can be described as we have a search agent which is searching for passengers and black boxes to save and return safely to stations and the problem have some attributes which are the agent which is the coast guard and it has a capacity of passengers and the grid which is containing the ships, the stations and the Coast guard boat. Every ship has some passengers and every ship has a black box which has a health that starts to decrease depending on time after all the passengers have died or been rescued and the ship is wrecked. For the stations, they are the return points where the Coast guard can safely return the passengers. So mainly our problem is to maximise the saved passengers and black boxes with our agent using different search algorithms like: BFS, DFS, iterative DFS, greedy and A*.

- A discussion of your implementation of the search-tree node ADT

For the node ADT, we have state, parent, leading action and depth. The state is another data type which contains some attributes which are the grid, the boat, the death, the destroyed boxes, the current damage of the damaged boxes and that total retrieves of passengers. So in this case, the state is representing the remaining ships, the boat object and the action leading to this state. In the node class, we also have the parent node which indicates the parent of this note which leads to it. And also we have the leading action which indicates the action happened from the parent to lead to this node. Last but not least, we have the depth of the node which indicates the level of the node in the search tree to be able to give it some heuristic which will be explained in detail in the next parts.

- A discussion of your implementation of the search problem ADT

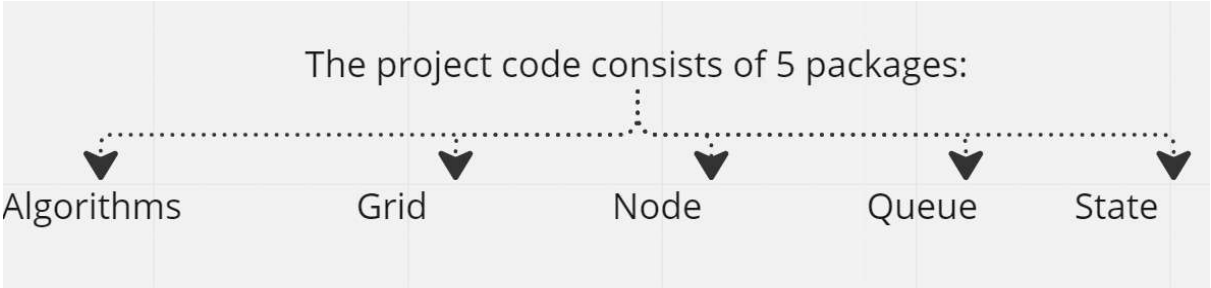
We have the general search abstract class and in this class we have some attributes which indicate the search problem. First, We have the initial state which is the root of the tree. Then we have the action list which contains actionType objects to indicate the series of actions in this search. We also have to check the goal test which will be overridden in the inheriting classes and also the path cost which will be also overridden in the inheriting classes. After that, we are going to discuss the Coast guard class which extends the general search abstract class.

- A discussion of your implementation of the CoastGuard problem

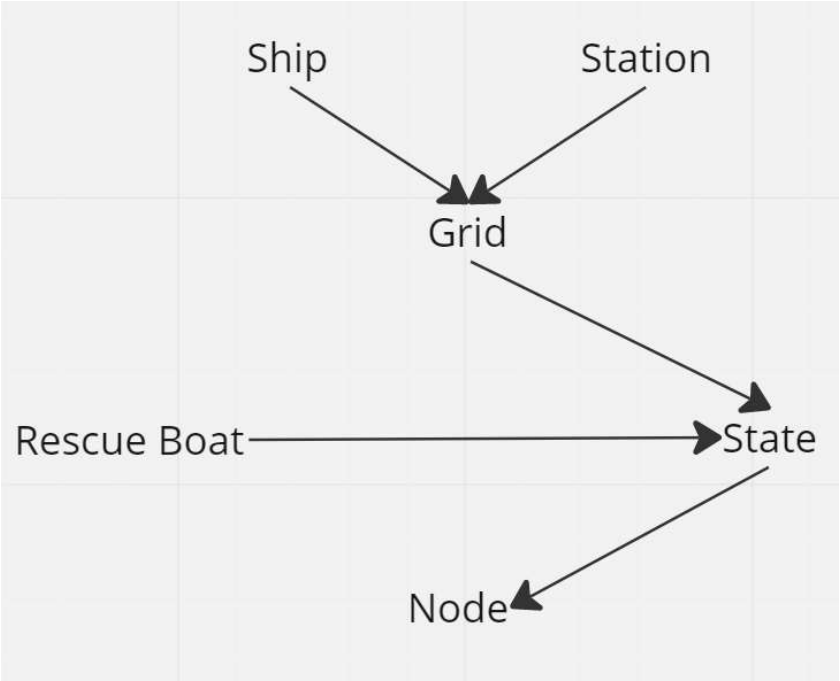
In this class(CoastGuard Class) we are extending the general search and override its methods to help indicate which search algorithm will be used. So we use the solve method to search a grid using the search class. The search class has a lot of attributes such as: strategy(the type of the search algorithm), The initial state, the action list, the previous

states, the number of expanded nodes and the visualise boolean which indicates if the grid would be shown on the console or not. And also it overrides the check goal test of the parent which checks if the grid has no ships and the boat is empty. Last but not least, we have the path cost method which gives the cost to every move on the grid to sort it according to the given search algorithm.

- A description of the main functions you implemented.



1) **Algorithms Package:**
   - **Search Class**, this class is responsible for applying different search strategies.
2) **Grid Package:**
   - **Grid Class**: this class represents the real grid, its size, the ships it has, the stations it has.
   - **GridGenerator Class**: contains the genGrid function which creates a random grid based on some specifications.
   - **RescueBoat class**: contains the rescue boat object logic, boat location, capacity, available cap, number of retrieved black boxes, number of saved people.
   - **Ship class**: represent the Ship and Wreck object logic , location, black box health, passengers.
   - **Station class**: represent the Station object, location, saved people and saved black boxes.
3) **Node Package:**
   - **Action Type Enum**: contains different actions/operators we have through the coast guard problem.
   - **Node Class:** contains the node object which has a State, parent, leading action and depth. It also contains the different actions that may be performed on a node. It also contains the different heuristics functions we have.
   - **Perform Action Class:** it takes an action and returns the node after we applied the action on it. IF the return is null, means we couldn't perform this action on the node.
4) **Queue Package:**
   - **Search Queue**: It is the best data type we have in our project. It has a global queue implementation based on a given strategy.
5) **State Package:**
   - **State Class:** contains the current State logic, the Grid state we have now, deaths, saved people, saved boxes, lost boxes.



This figure shows the links we have between our main objects, that the project is built on them.

We will go through each class in detail from small pieces until we reach how the final system works.

1) **Ship Class:**
   - **update()**: this function updates the ship status, by decreasing the passengers by 1, if they are 0 it becomes a wrecked one with a black box health 0, update it by 1 each time until 20, then it becomes fully damaged.
2) **Grid Class:**
   - **HashMap<String,Ship> shipsHashMap**: contains all ships(ship or wreck) we have.
   - **HashMap<String,Station> stationsHashMap**: contains all Stations we have,
   - **update():** loop over each ship we have and apply ship.update() and if the ship is fully damaged it removes it from shipsHashMap.
   - **calculateMinDistanceStation(RescueBoat b):** Calculates the nearest station to the boat based on Manhattan distance.
   - **calculateMinDistanceWreckedShip(RescueBoat rescueBoat)**: Calculates the nearest Wrecked Ship to the boat based on Manhattan distance.
   - **calculateMinDistanceNonWreckedShip(RescueBoat rescueBoat):** Calculates the nearest Ship to the boat based on Manhattan distance.
   - **getHighestNumOfPossibleSavedPassengers(RescueBoat rescueBoat):** get the highest possible saved passengers, by looping on every ship and remove the distance from its passengers.

3) **GridGenerator Class**:
   - **genGrid():** returns a random generated grid string.

- **buildGrid (String stringGrid)**: return an array of objects with size 2. [0] is the rescue boat object. [1] is the Grid object.

4) State Class:

- **update()**: it uses the update method inside the Grid and sets the num of deaths, damaged boxes of the current state.
- **gridVisualization()**: prints a whole grid visualisation of the current state.

5) Node Class:

- **right()**: performs right action.
- **left()** : performs left action.
- **up():** performs up action.
- **down():** performs down action.
- **pickup():** performs pickup action.
- **retrieve()**: performs retrieve action.
- **drop()**: performs drop action.
- **getActionsPath():** returns a string path that represents the path from the root until we reached this current node.
- getGoalTestNodeString(): returns the string that is needed in the project tests.
- **h1():** the heuristic that is used in GR1 and AS1
- **h2():** the heuristic that is used in AS2
- **h3():** the heuristic that is used in GR2
- **int aStar1()**: returns the path cost + h1(). Used in GR1
- **int aStar2():** returns the path cost + h2(). Used in GR2

6) PerformAction Class:

- **perform(Node node,String actionType)** : returns the result node from performing actionType on the given node.

7) SearchQueue Class:

- It generates the queue that will be used based on the given strategy, it works as a normal queue in case of BFS, as Stack in case of ID and DF, as PriorityQueue in Case of A* and GR.
- **add(Node node)**
- **poll()**
- **isEmpty()**
- **getComparableFunction()**: returns a comparator function in case of A* and GR.

8) Search Class:

- **Node search()**: A function that works for all search algorithms we have and returns a goal test node.

9) GeneralSearch Class:

- An abstract general Search Class, that has initialsState, list of actions, function to test the goal test and function to get the path cost.

10) **CoastGuard class:**
    - Extends GeneralSearch Class.
    - **solve (String grid, String strategy, boolean visualize)**: it returns the solution string based on a given strategy on a given grid.
    - **genActionList():** generate the actions/operators we have for the Coast Guard problem.
    - **genInitialState(String grid):** returns the initial state for the given grid.
    - **checkGoalTest(Object o):** overrides the checkGoalTest(Object o) we have in General Search Class.
    - **pathCost(Object o):** overrides the pathCost(Object o) that we have in General Search Class.

- A discussion of how you implemented the various search algorithms.

    - Here comes the beauty of The SearchQueue we implemented. The various algorithms are implemented in one method in **the Search Class.** Since the only difference between various algorithms is how its queue is implemented.
    - we have a root node push it to the queue and every time we check if it passes the goal test or not. Others we try to apply different actions we have on a copy of it and push it to the queue if we never faced such a node.

- A discussion of the heuristic functions you employed and, in the case of greedy or A*, an argument for their admissibility

In the first approach we used a heuristic function that prioritize the node that will lead to the nearest ship in order to save its passengers. If there is no case where there are no passengers it will look for the wrecked ships. Otherwise, there are no more ships on the grid and it should find a station.

```java
public int h1() {
    Grid grid = getState().getGrid();
    RescueBoat rescueBoat = getState().getBoat();
    int wreckedDistance = grid.calculateMinDistanceWreckedShip(rescueBoat);
    int nonWreckedDistance = grid.calculateMinDistanceNonWreckedShip(rescueBoat) ;
    int station = grid.calculateMinDistanceStation(rescueBoat);

    if(rescueBoat.isFull())
        return  station;
    if(nonWreckedDistance != Integer.MAX_VALUE) {
        return  nonWreckedDistance;
    }
    if(wreckedDistance != Integer.MAX_VALUE)
        return wreckedDistance;

    return -(grid.getM()+grid.getN()) +station ;
}
```


In the second heuristic function, instead of prioritizing the nearest non-wrecked ship. We gave priority to the ship that have more passengers onboard.

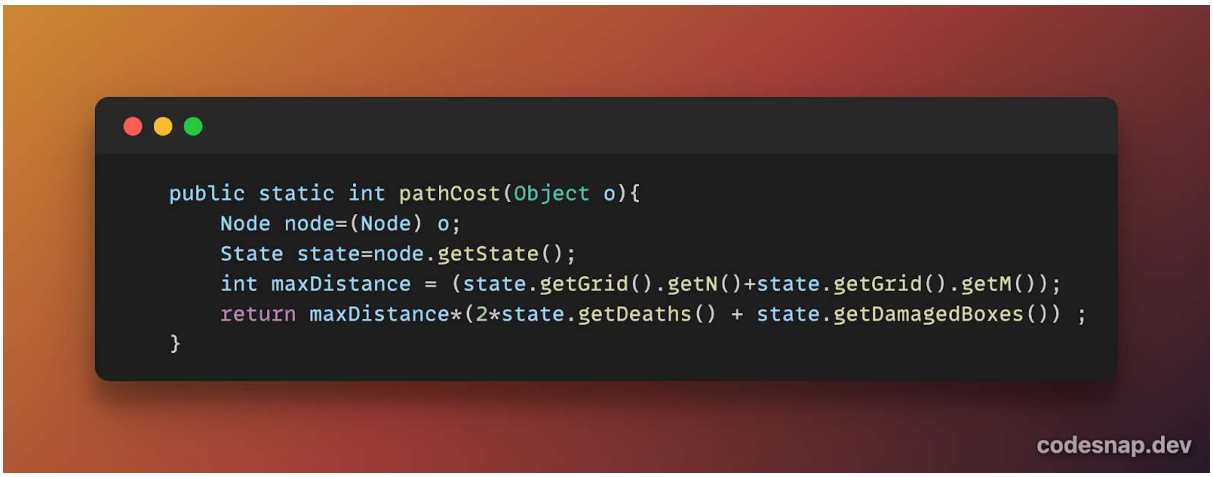
```java
public int h2() {

    Grid grid = getState().getGrid();
    RescueBoat rescueBoat = getState().getBoat();
    int possibleSavedPassengers = grid.getHighestNumOfPossibleSavedPassengers(rescueBoat);
    int wreckedDistance = grid.calculateMinDistanceWreckedShip(rescueBoat);
    int station = grid.calculateMinDistanceStation(rescueBoat);

    if(rescueBoat.isFull())
        return  station;
    if(possibleSavedPassengers == Integer.MIN_VALUE) {
        if (wreckedDistance != Integer.MAX_VALUE)
            return wreckedDistance;
    }
    return  possibleSavedPassengers;
}
```


For the path cost, we used the following function where we sum the number of death in the current state with the number of fully damaged boxes. It was also more convenient to give a higher weight to the deaths over the damaged boxes. To make sure that the heuristic functions does not over weight the path cost, we multiplied by the maximum distance M+N. We used another approach where we did not multiply by the maximum distance at neither the path cost or the heuristics but this approach worked better for our implementation. After multiplying by the maximum distance, the

heuristic funcion is still admissable as we add a distance to a ship and the cost of taking that path will increase by a larger amount of both death and boxes.



- A comparison of the performance of the different algorithms implemented in terms of completeness, optimality, RAM usage, CPU utilization, and the number of expanded nodes. You should comment on the differences in the RAM usage, CPU utilization, and the number of expanded nodes between the implemented search strategies

Algorithm	CPU %	RAM (MB)	Node Expansions	Deaths	Retrieves
BF	8.181818182	51.47363636	6983.090909	35.45454545	2.818181818
DF	7.181818182	3.834545455	94.18181818	123.1818182	1.363636364
ID	11.18181818	109.3490909	242602.4545	37.45454545	2.909090909
GR1	7.571428571	21.2575	2876.125	125.75	1.75
GR2	10.42857143	70.04142857	21540.71429	51.57142857	3.571428571
AS1	9	38.02142857	4465.714286	41.42857143	3.428571429
AS2	8.714285714	53.58428571	9717	44.14285714	3