Coast Guard Report

Team Name

AFs

Team Members

Aly Yasser Hegazy	T10	46-0371
Farida Samer Aldesouky	T10	46-0402
Farida Ashraf Othman	T17	46-0305

Table of Contents:

Discussion of the Problem:

The problem revolves around having a coast guard AI that uses different search algorithms to save people in the sea. The people are on ships and every action the coast guard takes, someone dies on every ship within the coast guard's radius (Grid). After all the passengers die, the ship is wrecked and the coast guard's mission is to retrieve the box it left behind, but the box expires in 20 actions (Time steps). The coast guard has a 4-drirectional movement capability and can pick up passengers and drop them off at stations, as well as retrieve boxes.

Formal Definition:

World: M X N Grid | that includes { I(1), I(2) , ... I(N) & S(1),S(2),....,S(N)}

Where I stands for stations and S stands for ships

Agent: The Coast Guard

Initial State: {G, Cgx, Cgy, C, S{S1, S2 ,...}, I{I1, I2, ...}}

G is the Grid
CgX, CgY coordinates
C Agent capacity
S Ships
I Stations

Operators: {Left, Right, Up, Down, Pickup, Drop, Retrieve}

Path Cost: estimated number of deaths and boxes lost per step.

Goal Test: {S{} && C == MaxC}

Classes Overview:

```java
public class Grid {

    private String[][] grid;

    private int M; // columns
    private int N; // rows

    private int maxC; //max capacity
    private int C; // capacity

    private int cgX; //coast guard x
    private int cgY; //coast guard y

    private int[][] i; //stations coordinates

    private ArrayList<Ship> s = new ArrayList<Ship>(); //ships coordinates

    private boolean gameOver = false;
    private int deaths = 0;
    private int savedPassengers = 0;
    private int collectedBoxes = 0;
```

The Grid class includes the world, meaning everything that happens or affects the world is being tracked in this class, further implementation will be discussed in section 3.

```java
public class Ship {
  private int remainingPassengers;
  private int boxDamage;
  private boolean boxRetrieved;
  private boolean wrecked;
  private boolean destroyed;
  private int x;
  private int y;
```

The Ship class includes the ships, that are referenced in the S ArrayList in the Grid class, this class is responsible for keeping track of individual ships as they change throughout the search.

Classes Overview:

```java
public class State {
    Grid grid;
    State parent;
    ArrayList<String> plan;
    String operator;
    int cgX;
    int cgY;
    int pathCost;
    int depth;
    int savedPassengers;
    int collectedBoxes;
    int deaths;
    int numberNodesExpanded;
    int heuristicValue;
```

The State class includes the Node implementation for the search trees generated, we don't need to literally generate a tree as this class's job is to simulate that. This class includes methods responsible for cloning states and keeping track of each of the important attributes as well as executing the next action, and keeping track of the plan within the current State (or node).

```java
public abstract class Search {
    private String[] operators;
    private State initState;
    private HashSet<State> stateSpace = new HashSet<State>();
    private ArrayList<State> queue = new ArrayList<State>();
    private String goalState;
```

The Search class acts the tree and the search, it is responsible for keeping track of the search problem, it uses an ArrayList for States to simulate a Tree, Queue or Stack (depending on the algorithm) and uses a hashSet for the state space to easily and efficiently eliminate all redundant states, and improve efficiency.

Implementations and Main Functions

Grid Functions:

```
    public static Grid genGrid() {
        //Generate between 5 and 15 locations
        int m = (int) ((Math.random() * (15 - 5)) + 5);
        int n = (int) ((Math.random() * (15 - 5)) + 5);

        Grid grid = new Grid(m, n);

        //Capacity between 30 and 100
        int c = (int) ((Math.random() * (100 - 30)) + 30);
        grid.setMaxC(c);
        grid.setC(c);

        //Coast Guard
        int cgX = (int) (Math.random() * m);
        int cgY = (int) (Math.random() * n);
        grid.setCgX(cgX);
        grid.setCgY(cgY);

        // Generate N stations between 1 and 40% of the grid size
        int[][] i = new int[(int) (Math.random() * (m * n) * 0.4) + 1][2];
        grid.setI(i);
        for (int j = 0; j < i.length; j++) {
            int x = (int) (Math.random() * m);
            int y = (int) (Math.random() * n);
            int[] station = {x, y};
            grid.getI()[j] = station;
            grid.getGrid()[y][x] = "Station";
        }


// Generate N ships between 1 and 40% of the grid size and each ship has a random number of passengers between
0 and 100
        for (int j = 0; j < (int) (Math.random() * (m * n) * 0.4) + 1; j++) {
            int x = (int) (Math.random() * m);
            int y = (int) (Math.random() * n);

            // If the ship is on a station, generate a new position
            while (grid.getGrid()[y][x] == "Station") {
                x = (int) (Math.random() * m);
                y = (int) (Math.random() * n);
            }

            int p = (int) (Math.random() * 100);
            Ship ship = new Ship(x, y, p);
            grid.getGrid()[y][x] = "Ship";
            grid.getS().add(ship);
        }

        return grid;
    }
```

Responsible for generating random Grids with random content.

Implementations and Main Functions

```
public String generateGridString()
```

```
    public static Grid decodeString(String grid)
```

Functions responsible for changing from String to class Grid and back.

```
1  public void dropOffPassengers()
```

```
1  public void pickUpPassengers()
```

```
public void moveAgent(String direction)
```

```
public void retrieveBox()
```

```
public void damageShips()
```

Functions responsible for executing operations within the Grid, dropoff checks if there are passengers and that the agent is at a station to drop,
pickup checks if the agent is on a ship that has passengers and has space to pick them up, retrieve checks if the agent is on a wreck and retrieves then destroys the ship by removing it from the ArrayList, damage ships is responsible for damaging all ships, if there are no passengers turning ship to wreck and destroying the ships if the box expires. this function is called every timestep.

Implementations and Main Functions

```java
public ArrayList<String> getPossibleActions() {
        ArrayList<String> actions = new ArrayList<String>();
        for (int i = 0; i < s.size(); i++) {
            if (s.get(i).getX() == cgX && s.get(i).getY() == cgY) {
                if (
                        s.get(i).getRemainingPassengers() > 0 &&
                                C > 0 &&
                                !s.get(i).isWrecked()
                ) {
                    actions.add("pickup");
                }
            }
        }
        if (grid[cgY][cgX] == "Station" && C < maxC) {
            actions.add("drop");
        }
        if (grid[cgY][cgX] == "Wreck") {
            for (int i = 0; i < s.size(); i++) {
                if (s.get(i).getX() == cgX && s.get(i).getY() == cgY) {
                    if (s.get(i).isWrecked() && !s.get(i).isDestroyed()) {
                        actions.add("retrieve");
                    }
                }
            }
        }
        if (cgY > 0) {
            actions.add("up");
        }
        if (cgX > 0) {
            actions.add("left");
        }
        if (cgX < M - 1) {
            actions.add("right");
        }
        if (cgY < N - 1) {
            actions.add("down");
        }

        //print all possible actions
//    System.out.println("Possible actions: ");
//    for (int i = 0; i < actions.size(); i++) {
//      System.out.print(actions.get(i) + " ");
//    }
//    System.out.println();
        return actions;
    }
```

This function is responsible for finding all possible actions executable within the following Grid, and is used to expand nodes are return all possible outcomes for a Tree, it returns a list of all possible actions which are later used as operators for the State. For example it loops over the ships to check if the agent is on any of them to add the pickup action, same for stations if the agent has passengers on, same for retrieve if the agent is on a wreck, and naturally it checks the possible directional movements and adds them. As seen in the code snippet the priority of actions is for the pickup, drop and retrieve actions as they have a bigger impact.

Implementations and Main Functions

```
public void performAction(String action)
```

Perform action takes a String and performs the action given as well as damageShips, this method is used to update the grid and prepare it for the next State (or step).

Copying Functions:

To avoid passing references and overwriting parent states, all the classes have their own cloning methods that basically copy all none-primitive data types and re-initializes them as a new instance to clone them, examples for that would be copying the Grid by looping over it, same for the ships ArrayList , etc..
After copying all the states, a new Instance of the object is initialized. Classes that implement the copying methods are: State, Grid and Ship.

Heuristic Calculation Functions:

There are 2 heuristics used, one that works by estimating number of deaths and trying to minimize the number of deaths by assigning a heuristic based on the number of deaths in the next time step, The other assigns a heuristic by estimating the number of boxes that would be lost in the next time step thus effectively maximizing the number of boxes collected; this will be discussed in detail in the heuristic in this section we will only discuss specific functions that are used.

The heuristics are calculated in 4 steps:

1. Run ClosestShip() or ClosestBox() depending on the heuristic, each of those methods returns the index of the closest ship or box to the agent, and are run at each step to update the index.
2. Run BestActionToShip() which uses NextActionToShip() iteratively to generate the best path of actions to the current objective.
3. The heuristic of the next state is assigned by comparing the operator of the state to the the index of that operator in the best path returned from step 2, if the operator is not in the best action it is assigned a higher heuristic EX best = [right, left] if a state has right it is assigned a heuristic of 0 (The lowest possible), left 1, anything else would have a heuristic of 3 (unless action is pick, drop, retrieve).
4. The lowest heuristic is then picked and expanded. In case its a tie, random state is expanded.

Implementations and Main Functions

Coast Guard Helpers:

```java
public State expand(State prevState) {
        String action = prevState.getOperator();
        //System.out.println("Expanding: " + action);

        //init the state
        Grid currentGrid = prevState.getGrid();
        ArrayList<String> plan = prevState.getPlan();
        int cgX;
        int cgY;
        int pathCost = prevState.getPathCost();
        int depth = prevState.getDepth();
        int numberNodesExpanded = prevState.getPlan().size();


        //perform the action
        currentGrid.performAction(action);
        int newDeaths = currentGrid.getDeaths();
        int newSavedPassengers = currentGrid.getSavedPassengers();
        int newCollectedBoxes = currentGrid.getCollectedBoxes();

        //update the state
        plan.add(action);
        pathCost++;
        numberNodesExpanded++;
        cgX = currentGrid.getCgX();
        cgY = currentGrid.getCgY();


        State newState = new State(
                currentGrid,
                prevState,
                plan,
                action,
                cgX,
                cgY,
                pathCost,
                depth,
                newSavedPassengers,
                newCollectedBoxes,
                newDeaths,
                numberNodesExpanded,
                prevState.getHeuristicValue()

        );

        return newState;
    }
```

The Expand function works by performing actions on the current state and generating the next state, of course the state itself is copied in respective search algorithm.

Implementations and Main Functions

```java
public int calculatePathCost() {
        int pathCost = 0;
        State state = this;
        Grid grid = state.getGrid();
        for(Ship ship : grid.getS()) {
            if(!ship.isWrecked()) {
                pathCost += ship.getRemainingPassengers();
            }
            else if (ship.isWrecked() && ship.getBoxDamage() == 19) {
                pathCost ++;
            }
        }
        return pathCost;
    }
```

```java
    public static String stringifyState(State node) {
        ArrayList<String> plan = node.getPlan();
        //print the plan
        String planString = "";
        for (String action : plan) {
            planString += action + ",";
        }
        //System.out.println("Plan size: " + plan.size());
        planString = planString.substring(0, planString.length() - 1);
        int deaths = node.getDeaths();
        int retrieved = node.getCollectedBoxes();
        int nodes = node.getNumberNodesExpanded();
        return planString + ";" + deaths + ";" + retrieved + ";" + nodes;
    }
```

Calculating path cost for A* works by predicting the deaths and boxes lost each step, and the StringifyState simply returns the solution format.

Search Algorithms

Depth First Search:

The depth first search for the Coast Guard is linear; in the sense that since any taken branch in the tree would lead to a solution, the DFS is complete, thus linear in the sense that taking any single branch and going straight down would lead to a solution, since all the passengers would eventually be rescued or die, and even the rescued ones would eventually be dropped off; moreover, all the ships would turn to wrecks at some point and all the boxes would either be collected or expire, and that means that any randomly generated branch would lead to a goal state, given of course that it is not stuck in an infinite loop, how so?
Imagine picking up passengers then starting to move [Left, Right, Left, ...] as the agent gets stuck in a corner, never being able to drop off passengers thus never fulfilling the goal state.
Fixing that problem and maintaining the linearity of the agent to optimize performance was as simple as randomizing motion for the agent (given of course that there is always priority to picking up, dropping off and retrieving, meaning that no randomization occurs if the branch leads to those actions). This guaranteed a depth first search that always reached a goal state, although it is non-deterministic due to the motion randomization. This of course is how depth first works given that the branch is generated randomly and procedurally as the Agent expands the different States. That means that the DFS for Coast Guard is complete, of course it is not optimal and it has a time complexity of O(N * S) where S is the number of ships since they are checked every time, but given that the number of ships is small O(N).

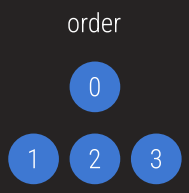
```
while (!currentGrid.checkGameOver()) {
            ArrayList<String> actions = currentGrid.getPossibleActions();
            String nextOperator = actions.get(0);
            //if the top action is a move, then randomly choose a move
            if (
                    nextOperator.equals("left") ||
                            nextOperator.equals("right") ||
                            nextOperator.equals("up") ||
                            nextOperator.equals("down")
            ) {
                nextOperator = actions.get((int) (Math.random() * actions.size()));
            }
            currentState.setOperator(nextOperator);
            State nextState = searchProblem.expand(currentState);
            currentState = nextState;
            currentGrid = currentState.getGrid();
            searchProblem.getQueue().add(currentState);
            searchProblem.getStateSpace().add(currentState);
```

Search Algorithms

Breadth First Search:

The Breadth First Search is more complex than the DFS as this time it is not linear, all states have to be expanded to form a tree-like structure and the states must be placed in their proper order to follow the branch order, that was done by using a Queue to always know what the next action is, and when the next action is expanded the States were then enqueued at the back of the Queue using an ArrayList in their proper location.

Example of the Queue:



A expand A

A1, A2 , A3 expand A1

A2,A3,A11,A12,A13 expand A2

when expanding, as the example shows the State is removed from the Queue and replaced with the expansion results, where the expansion results are generated using getPossibleActions.

Another challenge for BFS is redundant States as redundant States could lead to expanding the same tree more than once, drastically increasing the time and space complexity, to avoid this problem a HashSet was used for StateSpace where the hashCode was cgX, cgY, operator and deaths, as those attributes if repeated would result in a whole tree being repeated, and that would create redundant States. A similar equals method was created to guarantee that states in stateSpace only occur once. then if a State was already in the StateSpace it would automatically be discarded.

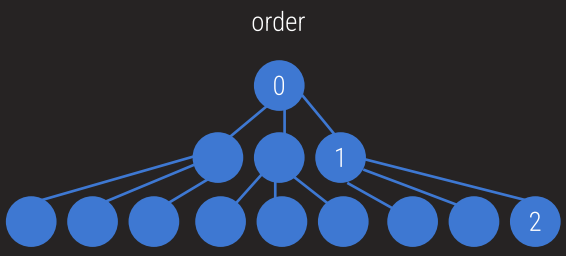
For the Coast Guard BFS is also complete (as are all the algorithms) and the complexity is O(N).

```
while (!currentGrid.checkGameOver()) {
            currentState = searchProblem.expand(searchProblem.getQueue().remove(0));
            currentGrid = currentState.getGrid();
            actions = currentGrid.getPossibleActions();
            //get possible actions
            for (String action : actions) {
                State temp = currentState.copy();
                temp.setOperator(action);
                if (!searchProblem.getStateSpace().contains(temp)) {
                    searchProblem.getQueue().add(temp);
                    searchProblem.getStateSpace().add(temp);
                }
            }
```

Search Algorithms

Iterative Deepening:

The Iterative Deepening works just like a DFS but this time traverses all the different branches just like the BFS, and resets as the currentDepth is reached, so this time all the branches in the DFS are expanded. when currentDepth is reached the depth is increased and the Iterative Deepening starts over effectively resetting the problem but with higher depth, until a goal state is reached. this time instead of dequeuing from the front of the ArrayList like the BFS we dequeue from the back to follow the pattern of the DFS.



Example at Depth = 3:

A expand A

A1, A2 , A3 expand A3

A1,A2,A31,A32,A33 expand A33

The Iterative Deepening is complete, and has a time complexity of $O(N^M)$ and a space complexity of O(NM)

```
while (!currentGrid.checkGameOver()) {
            if (increaseDepth) {
                currentGrid = grid.copyGrid();
                currentState = new State(
                        currentGrid,
                        null,
                        new ArrayList<String>(),
                        "Init State",
                        currentGrid.getCgX(),
                        currentGrid.getCgY(),
                        0,
                        0,
                        0,
                        0,
                        0,
                        0,
                        0
                );
                searchProblem.getQueue().clear();
                searchProblem.getStateSpace().clear();
                searchProblem.setInitState(currentState);
                searchProblem.getStateSpace().add(currentState);
                actions = currentGrid.getPossibleActions();
                for (String action : actions) {
                    State temp = currentState.copy();
                    temp.setOperator(action);
                    searchProblem.getQueue().add(temp);
                    searchProblem.getStateSpace().add(temp);
                }
            }
            increaseDepth = false;
            if (currentState.getDepth() <= depthLimit) {
                currentState = searchProblem.expand(searchProblem.getQueue().remove(
                        searchProblem.getQueue().size() - 1)
                );
                currentGrid = currentState.getGrid();
                actions = currentGrid.getPossibleActions();
                //get possible actions
                for (String action : actions) {
                    State temp = currentState.copy();
                    temp.setOperator(action);
                    if (!searchProblem.getStateSpace().contains(temp)) {
                        searchProblem.getQueue().add(temp);
                        searchProblem.getStateSpace().add(temp);
                    }
                }
```

Search Algorithms

Greedy & Heuristic:

The Greedy algorithm works by assigning a heuristic and then choosing which node to expand based on the lowest heuristic value for each State. There are two heuristics, one that estimates deaths, by finding the closest ship to save passengers from, and the other estimates boxes lost by estimating the closest wrecks and assigning the heuristic based on that. That makes the Greedy algorithm run in O(N) since it never needs to expand nodes that don't have the lowest heuristic cost, moreover upon tying a random node is expanded, and it's obviously complete since it always leads to a goal state, and it is optimal since it always does what the heuristic criteria is.

Heuristic 1 Deaths:

The first heuristic works by always finding the closest ship and assigning the heuristic based on how close to a ship effectively minimizing the number of deaths by always prioritizing pickups. When full, the lowest heuristic goes to dropping passengers off, thus saving the most number of lives and minimizing deaths. This heuristic also satisfies the centering property as it has H(N) = 0 on goal states, and on states where those states would eminently lead to a goal state. This function is admissible as it never overestimates the goal by always giving each state a heuristic based on whether or not this state leads to a goal state, and the assigning is also weighted, to make sure it never over estimates the cost.

Heuristic 2 Boxes:

The second heuristic works by always finding the closest wreck and assigning the heuristic based on how close to a wreck effectively minimizing the number of boxes lost by always prioritizing retrieves, thus saving the most number of boxes and minimizing box loss (eliminating it). Whenever there are no wrecks it operates similar to a DFS since all weights are similar. This heuristic also satisfies the centering property as it has H(N) = 0 on goal states, and on states where those states would eminently lead to a goal state. This function is admissible as it never overestimates the goal by always giving each state a heuristic based on whether or not this state leads to a goal state, and the assigning is also weighted, to make sure it never over estimates the cost.

Search Algorithms

A* & Cost function:

A* works by combining heuristics with a cost function, where the cost function is the addition of the number of deaths and the boxes lost in the next step, the path cost accumulates as the agent moves and during the cloning process it is transferred from one state to the next while adding the future loss, it then adds the heuristic to said cost function and navigates to the lowest cost State. This effectively results in the most optimal solution, it also runs in O(N) since it never needs to visit nodes that don't have the lowest cost, and of course is complete since it always leads to a goal state.

Performance Comparison

RAM Usage:

To compare the Ram usages of the different strategies, their RAM usages for test grids 0, 1, and 9 were calculated as follows:

Memory Usage for BF Test 0 : 1.633 MB, Test 1 : 10.947 MB, Test 9 : 102.196 MB

Memory Usage for DF Test 0 : 0 MB, Test 1 : 1.76 MB, Test 9 : 0.76 MB

Memory Usage for ID Test 0 : 1.33 MB, Test 1 : 27.82 MB, Test 9 : 3.44 MB

Memory Usage for GR1 Test 0 : 0 MB, Test 1 : 0 MB, Test 9 : 1.048 MB

Memory Usage for GR2 Test 0 : 0 MB, Test 1 : 1.048 MB , Test 9 : 1.633MB

Memory Usage for AS1 Test 0 : 0 MB ,Test 1 : 0 MB ,Test 9 : 0 MB

Memory Usage for AS2 Test 0 : 1.048 MB ,Test 1 : 0 MB ,Test 9 : 0 MB

As shown, BFS and ID take up more RAM than the other strategies while the strategies using the heuristics take up significantly less RAM due to A* being an optimally efficient strategy.

CPU Utilisation:

To compare the Cpu utilization of the different strategies, the difference in CPU percentages before and after applying the strategies was calculated for grid 5 as follows:

BF: 20 %

DF: 30 %

ID: 52 %

GR1: 31%

GR2: 51 %

AS1: 48 %

AS2: 41%

Nodes Expanded:

To compare the Node Expansion of the different strategies, their nodes expanded for test grids 0, 1, and 9 were calculated as follows:

Nodes Expanded for BF Test 0 : 4039, Test 1 : 2288, Test 9 : 27253

Nodes Expanded for DF Test 0 : 59, Test 1 : 38, Test 9 : 388

Nodes Expanded for ID Test 0 : 6179, Test 1 : 913, Test 9 : 6606

Nodes Expanded for GR1 Test 0 : 18, Test 1 : 12, Test 9 : 37

Nodes Expanded for GR2 Test 0 : 109, Test 1 : 38, Test 9 : 113

Nodes Expanded for AS1 Test 0 : 18 ,Test 1 : 12 ,Test 9 : 37

Nodes Expanded for AS2 Test 0 : 20 ,Test 1 : 15 ,Test 9 : 37

As expected the BFS expands a large number of nodes due to it having to go down different branches, and since in our problem the DFS continuing to go down one branch always leads to a solution it expands a drastically smaller number of nodes, the iterative deepening is mostly expanding nodes more than the BFS (due to it having to start over), but in one instance due to reaching a goal sate early it expanded less nodes as seen in test 1, finally both the heuristics in greedy lead to a small number of nodes expanded, although the one prioritizing saving passengers did better due to passengers having a larger impact, we could even say that it resulted in an optimal solution. Finally as expected the A* always gave an optimal solution and expands less nodes based on the heuristic of course.