Generic Search Problem Implementation:

Class Generic Search Problem is implemented as explained in Class, however abstract functions were not implemented in this class.

1- ArrayList<operator> operators //list of possible operators **//Enum {Up, Down, Left, Right, Retrieve, Drop, Pickup}.**

2- state initialState **//initial state of the problem.**

```
5  public class GenericSearchProblem {
6      ArrayList<operator> operators = new ArrayList<operator>();
7      state initialState;
8      //abstract Boolean GoalTest(node node);
9      //abstract int pathCost(node node);
10     //abstract node General_Search_Procedure(node initialNode, String strat);
11 }
12
```

Node State Implementation:

Class state is implemented to host the needed information for each node's expansion.

1- int x, y **//current location of Node/CoastGuard.**

2- int unrescuedPassengers **//number of remaining Passengers at current node state**

3- int deaths **//Total deaths at current node state**

4- int carriedPassengers **//coastguard carriedPassengers at current node state**

5- int undamagedBoxes **//total number of undamagedBoxes remaining at current node state**

6- int retrievedBoxes **//total No. Of retrievedBoxes at current node state**

7- int lostBoxes **//total no. of lost boxes at current node state**

8- ArrayList<Ship> ships **//list of ships & their states at current node state**

```java
public class state {
    //CoastGuard
    int x;
    int y;
    //Passengers
    int unrescusedPassengers;
    int deaths;
    int carriedPassengers;
    //Boxes
    int undamagedBoxes;
    int retrievedBoxes;
    int lostBoxes;
    //currentShips
    ArrayList<ship> ships = new ArrayList<ship>();
```

Search Tree Node Implementation:

Class Node is implemented as introduced in class.

1- Node parentNode **//parent node of current node.**
2- operator operator **//Enum {Up, Down, Left, Right, Retrieve, Drop, Pickup}.**
3- int depth **//depth of the node.**
4- Int [ ] pathCost **// Tuple consisting of {state.deaths, state.lostBoxes} which will be used in UCS and A*.**

5- state state **//state of current node.**

```
public class node{
    node parentNode;
    operator operator;
    int depth;
    int[] pathCost = new int[2];
    state state;
    String Solution;
    public node(node parentNode, operator operator,
            int depth, int[] pathCost, state state) {
        this.parentNode = parentNode;
        this.operator = operator;
        this.depth = depth;
        this.pathCost = pathCost;
        this.state =  state;
    }


}
class UCScomparable implements Comparator<node>{
```

Coast Guard Implementation:

This is the main class that hosts takes a node and checks if there is an existing black box on this node or not.

Global variables:

1- int gridM, gridN **//M, N bounds of grid**

2- int maxPassengers **//max passengers**

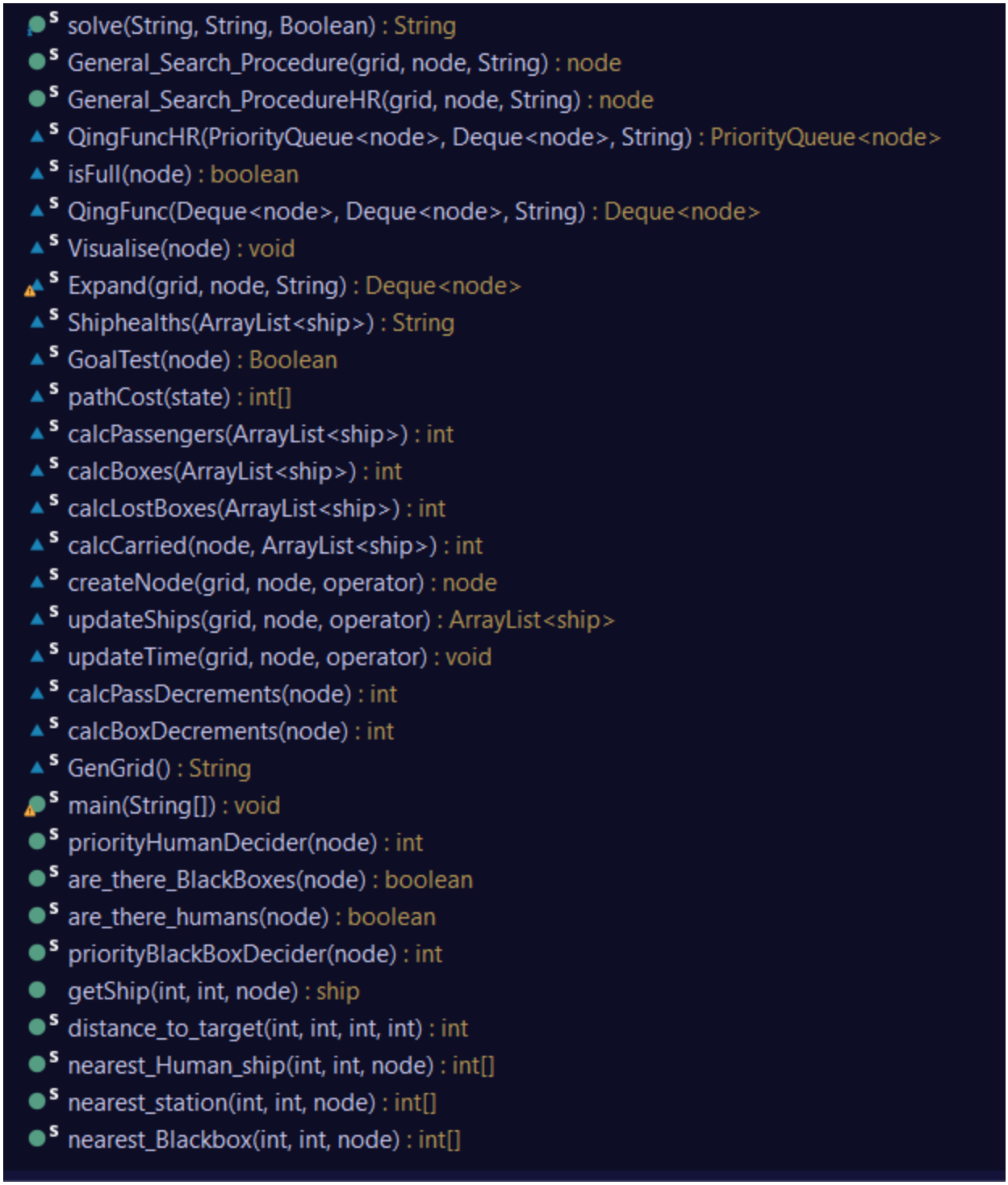
3- ArrayList<station> StationsList **//list of stations to be used globally for actions**

4- int iterativeDepth **//global variable of depth to be used for IDS**

5- HashMap<String, state> RepeatedStates **//global HashMap to keep track of RepeatedStates.**

The problem returns the solution through function Solve ( ), which creates the initial node, which is essentially the general search algorithm introduced in class.
Which creates an initial node, and then expands on its children and enqueues them according to the enqueuing strategy, and dequeues those nodes until it finds a solution.

```java
public class CoastGuard extends GenericSearchProblem {


    static int gridM = 0;
    static int gridN = 0;
    static int maxPassengers = 0;
    static ArrayList<station> stationslist = new ArrayList<station>();
    static int iterativeDepth = 0;
    static HashMap<String, state> RepeatedStates;

```


solve(String, String, Boolean) : String
General_Search_Procedure(grid, node, String) : node
General_Search_ProcedureHR(grid, node, String) : node
QingFuncHR(PriorityQueue<node>, Deque<node>, String) : PriorityQueue<node>
isFull(node) : boolean
QingFunc(Deque<node>, Deque<node>, String) : Deque<node>
Visualise(node) : void
Expand(grid, node, String) : Deque<node>
Shiphealths(ArrayList<ship>) : String
GoalTest(node) : Boolean
pathCost(state) : int[]
calcPassengers(ArrayList<ship>) : int
calcBoxes(ArrayList<ship>) : int
calcLostBoxes(ArrayList<ship>) : int
calcCarried(node, ArrayList<ship>) : int
createNode(grid, node, operator) : node
updateShips(grid, node, operator) : ArrayList<ship>
updateTime(grid, node, operator) : void
calcPassDecrements(node) : int
calcBoxDecrements(node) : int
GenGrid() : String
main(String[]) : void
priorityHumanDecider(node) : int
are_there_BlackBoxes(node) : boolean
are_there_humans(node) : boolean
priorityBlackBoxDecider(node) : int
getShip(int, int, node) : ship
distance_to_target(int, int, int, int) : int
nearest_Human_ship(int, int, node) : int[]
nearest_station(int, int, node) : int[]
nearest_Blackbox(int, int, node) : int[]

Main functions:

1- General_Search_Procedure

This is the general search algorithm introduced in class, which takes the initialNode which is created in Solve ( ), and expands the currentNode thru Expand() function, and enqueues them according to the enqueueing strategy thru QingFunc.

//works only for uninformed algorithms

//General_Search_ProcedureHR() works for heuristics

```
public static node General_Search_Procedure(grid grid, node initialNode, String strat) {
    Deque<node> nodes = new LinkedList<>();
    nodes.add(initialNode);
    while (!nodes.isEmpty()) {
        node currNode = nodes.removeFirst();
        if(GoalTest(currNode) == true) {
            System.out.println("Success");
            return currNode;
        }
        Deque<node> newNodes = new LinkedList<>();
        newNodes = Expand(grid, currNode, strat);
        nodes = QingFunc(nodes, newNodes, strat);
        // System.out.println("GS expanded: " + nodes.size());

    }
    System.out.println("Failure");
    return null;

}
```

2– Expand()

This is the expand algorithm which takes a node, and looks into the possible actions it can do, and adds it to a list (to be later enqueued by QingFunc( ))

```
static Deque<node> Expand(grid grid, node node, String strat) {

    Deque<node> nodes = new LinkedList<>();
    if (grid.possiblemove(operator.Drop, node) == true) {

        nodes.addFirst(createNode(grid, node, operator.Drop));

    }
    if (grid.possiblemove(operator.Pickup, node) == true) {

        nodes.addFirst(createNode(grid, node, operator.Pickup));

    }
    if (grid.possiblemove(operator.Retrieve, node) == true) {

        nodes.addFirst(createNode(grid, node, operator.Retrieve));

    }

    if (grid.possiblemove(operator.Right, node) == true) {

        node newNode = createNode(grid, node, operator.Right);
        String key = Integer.toString(newNode.state.x) + Integer.toString(newNode.state.y)
                + Integer.toString(newNode.state.unrescusedPassengers) + Integer.toString(newNode.state.deaths)
                + Integer.toString(newNode.state.undamagedBoxes) + Integer.toString(newNode.state.retrievedBoxes)
                + Shiphealths(newNode.state.ships) + "";
        //if(!RepeatedStates.containsKey(key)|| strat == "GR1") {
        if (!RepeatedStates.containsKey(key)) {
            nodes.addFirst(newNode);
            RepeatedStates.put(key, newNode.state);
        }
    }
```

3– QingFunc()

This is the queueing function that updates the Nodes queue in the General Search Algorithm according to the enqueuing strategy.
//works for non heuristic algorithms

```java
static Deque<node> QingFunc(Deque<node> nodes, Deque<node> newNodes, String strat) {
    // Deque<node> finalNodes = nodes;
    Deque<node> finalNodes = new LinkedList<>();

    switch (strat) {
    case "DF":
        finalNodes.addAll(nodes);
        for (int i = 0; i < newNodes.size(); i++) {
            finalNodes.addFirst(newNodes.removeFirst());
        }
        break;
    case "BF":
        finalNodes.addAll(nodes);
        for (int i = 0; i < newNodes.size(); i++) {
            finalNodes.addLast(newNodes.removeFirst());
        }
        break;
    case "ID":
        finalNodes.addAll(nodes);
        for (int i = 0; i < newNodes.size(); i++) {
            node tempNode = newNodes.removeFirst();
            if (tempNode.depth <= iterativeDepth) {
                finalNodes.addFirst(tempNode);
            }
        }
        break;
    default:
        break;
    }

    return finalNodes;
}
```

3– QingFuncHR()

This is the queueing function that updates the Nodes priority in the General Search Algorithm HR according to the enqueuing strategy.
//works for heuristic algorithms

```java
static PriorityQueue<node> QingFuncHR(PriorityQueue<node> nodes, Deque<node> newNodes, String strat) {
    PriorityQueue<node> finalNodes= new PriorityQueue<node>();
    switch (strat) {
    case "UC":
        finalNodes = new PriorityQueue<node>(new UCScomparable());
        finalNodes.addAll(newNodes);
        PriorityQueue<node> pq = new PriorityQueue<node>(new UCScomparable());
        pq.addAll(nodes);
        for (int i = 0; i < newNodes.size(); i++) {
            pq.add(newNodes.removeFirst());
        }
        finalNodes.addAll(pq);
        break;
    case "GR1":
        finalNodes = new PriorityQueue<node>(new GR1comparable());
        finalNodes.addAll(newNodes);
        PriorityQueue<node> pq1 = new PriorityQueue<node>(new GR1comparable());
        for (int i = 0; i < newNodes.size(); i++) {
            pq1.add(newNodes.removeFirst());
        }
        finalNodes.addAll(pq1);
        break;
    case "GR2":
        finalNodes = new PriorityQueue<node>(new GR2comparable());
        finalNodes.addAll(newNodes);
        PriorityQueue<node> pq2 = new PriorityQueue<node>(new GR2comparable());
        for (int i = 0; i < newNodes.size(); i++) {
            pq2.add(newNodes.removeFirst());
        }
        finalNodes.addAll(pq2);
        break;
```

4- CreateNode ( )

This function creates the expanded nodes, which are called in the Expand ( ) method, which also updates the node informations and time steps.

```
static node createNode(grid grid, node node, operator operation) {

    // int BoxDecrements = calcBoxDecrements(node);
    ArrayList<ship> updatedShips = updateShips(grid, node, operation);
    state newState = null;
    node newNode = null;
    int newPassengers = calcPassengers(updatedShips);
    int PassDecrements = calcPassDecrements(node);
    //int deaths = calcPassengers(grid.shipslist) - newPassengers;
    int deaths = node.state.deaths + PassDecrements;
    int x = node.state.x;
    int y = node.state.y;

    switch (operation) {
    case Up:
        newState = new state(x - 1, y, newPassengers, deaths, node.state.carriedPassengers, calcBoxes(updatedShips),
                node.state.retrievedBoxes,calcLostBoxes(updatedShips), updatedShips);
        newNode = new node(node, operator.Up, node.depth + 1, pathCost(newState), newState);
        newNode.Solution = node.Solution + "up,";
        break;
    case Down:
        newState = new state(x + 1, y, newPassengers, deaths, node.state.carriedPassengers, calcBoxes(updatedShips),
                node.state.retrievedBoxes,calcLostBoxes(updatedShips), updatedShips);
        newNode = new node(node, operator.Down, node.depth + 1, pathCost(newState), newState);
        newNode.Solution = node.Solution + "down,";
        break;
    case Left:
        newState = new state(x, y - 1, newPassengers, deaths, node.state.carriedPassengers, calcBoxes(updatedShips),
                node.state.retrievedBoxes, calcLostBoxes(updatedShips),updatedShips);
        newNode = new node(node, operator.Left, node.depth + 1, pathCost(newState), newState);
        newNode.Solution = node.Solution + "left,";
        break;
    case Right:
        newState = new state(x, y + 1, newPassengers, deaths, node.state.carriedPassengers, calcBoxes(updatedShips),
                node.state.retrievedBoxes,calcLostBoxes(updatedShips), updatedShips);
```

Helper functions

1. are_there_BlackBoxes

This Function takes a node and checks if there is an existing black box on this node or not.

```
public boolean are_there_BlackBoxes(node thisnode) {

    for (int i = 0; i < thisnode.state.ships.size(); i++) {
        if (thisnode.state.ships.get(i).hasBlackBox) {
            return true;
        }

    }

    return false;
}
```

2. Are_there_humans

This function takes a node and checks if there are humans on a ship on that node or not.

```
public static boolean are_there_humans(node thisnode) {

    for (int i = 0; i < thisnode.state.ships.size(); i++) {
        if (thisnode.state.ships.get(i).are_there_people_here()) {
            return true;
        }

    }

    return false;
}
```

3. getShip

This function takes a node and X and Y position on a grid and checks if there is a certain ship then it returns it's current state if there is no existing ship on that position it returns null.

```
public ship getShip(int x, int y, node thisnode) {
    for (int i = 0; i < thisnode.state.ships.size(); i++) {
        if (thisnode.state.ships.get(i).x == x && thisnode.state.ships.get(i).y == y) {
            return thisnode.state.ships.get(i);
        }
    }
    return null;
}
```

3. Distance_to_target

This function calculates the distance between the coast guard and any target (ships,stations,wrecks) by taking the position of the coast guard and the position of the desired target and returning the total distance.

```
public static int distance_to_target(int PlayerX, int PlayerY, int TargetX, int TargetY) {
    int total = 0;
    int x = PlayerX - TargetX;
    int y = PlayerY - TargetY;
    x = Math.abs(x);
    y = Math.abs(y);
    total = x + y;
    return total;

}
```

4. Possible Move

Possible move takes A node and an operator and it checks the feasibility of doing this operator with the data in the node.

```
if (operating == operator.Pickup) {

    int ShipNumber1 = isThisAShip(node);
    if (ShipNumber1 != -1) {
        if (node.state.ships.get(ShipNumber1).numberOfPassengers > 0
                && node.state.ships.get(ShipNumber1).isWreck == false
                && ((this.maxNumberofPassengers - node.state.carriedPassengers) > 0)) {
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }

}

if (operating == operator.Drop) {

    if (node.state.carriedPassengers > 0 && isThisAStation(node)) {

        return true;
    } else {

        return false;
    }

}
if (operating == operator.Retrieve) {

    // get function to check the amount of passengers on board\

    int stationNumber = isThisAShip(node);
    if (stationNumber != -1) {
        if (node.state.ships.get(stationNumber).hasBlackBox == true
                && node.state.ships.get(stationNumber).isWreck == true
                && node.state.ships.get(stationNumber).BlackBoxHp < 20) {
            return true;
        }
    } else {
        return false;
    }
```

For the Pickingup, Dropping off and retrieving, the code checks if the x and y positions are at a station/ship and whether the ship has boxes/humans on it.

```
        if (operating == operator.Up) {
            // if(y-1>0) {
            // if(node.state.x-1>=0 && node.state.x-1<M) {
            if (node.state.x - 1 >= 0) {
                return true;
            }
        }
        if (operating == operator.Down) {
            // if(y+1<N) {
            // if(node.state.x+1>0 && node.state.x+1<=M) {
            if (node.state.x + 1 < N) {
                return true;
            }
        }
        if (operating == operator.Right) {
            // if(x+1<M) {
            if (node.state.y + 1 < M) {
                return true;
            }
        }
        if (operating == operator.Left) {
            // if(x-1>0) {
            // if(node.state.y-1>0 && node.state.y-1<=N) {
            if (node.state.y - 1 >= 0) {
                return true;
            }
        }

        return false;
    }
```

For the Directions it checks the map size to whether it can go there or not

5. Is this a ship/station

```
    public int isThisAShip(node node) {
        for (int i = 0; i < node.state.ships.size(); i++) {
            if (node.state.ships.get(i).x == node.state.x && node.state.ships.get(i).y == node.state.y) {
                return i;
            }
        }
        return -1;
    }

    public boolean isThisAStation(node node) {
        for (int i = 0; i < stationslist.size(); i++) {
            if (stationslist.get(i).x == node.state.x && stationslist.get(i).y == node.state.y) {
                return true;
            }
        }
        return false;
    }

}
```

These functions takes a node and checks whether the coast guard is on a station/ship

6. Grid

```
public grid(String input) {
    String[] temp = input.split(";");
    // Now The String is split into Three Different sectors and should look like
    // this
    // [m,n; C ; grid] m and n need to be split again
    // [0 , 1 , 2]

    // M and N Extraction
    String[] MandN = temp[0].split(",");
    M = Integer.parseInt(MandN[0]);
    N = Integer.parseInt(MandN[1]);

    maxNumberofPassengers = Integer.parseInt(temp[1]);

    // initial coast guard location
    String[] coastGuardloc = temp[2].split(",");
    coastGuardX = Integer.parseInt(coastGuardloc[0]);
    coastGuardY = Integer.parseInt(coastGuardloc[1]);

    // Grid Extraction
    String[] stations = temp[3].split(",");
    for (int i = 0; i < stations.length; i += 2) {
        int x = Integer.parseInt(stations[i]);
        int y = Integer.parseInt(stations[i + 1]);
        station thisStation = new station(x, y);
        stationslist.add(thisStation);
    }
    String[] ships = temp[4].split(",");
    for (int i = 0; i < ships.length; i += 3) {
        int x = Integer.parseInt(ships[i]);
        int y = Integer.parseInt(ships[i + 1]);
        int Pass = Integer.parseInt(ships[i + 2]);
        ship thisShip = new ship(x, y, Pass, 1, false, true);
        shipslist.add(thisShip);

    }

}
```

Grid takes the string input and takes the values within it in
It and putting them into global variables of the instantiation.

7. Visualise

```
static void Visualise(node theNode) {
    String[][] map = new String[gridN][gridM];
    for (int i = 0; i < theNode.state.ships.size(); i++) {
        ArrayList<ship> currships = theNode.state.ships;
        ship thisShip = currships.get(i);

        if (!thisShip.isWreck) {
            map[thisShip.y][thisShip.x] = "Ship " + "(" + thisShip.numberOfPassengers + ")";

        } else {
            map[thisShip.y][thisShip.x] = "Wreck " + "(" + thisShip.BlackBoxHp + ")";

        }
    }

    for (int i = 0; i < stationslist.size(); i++) {
        station thisStation = stationslist.get(i);
        map[thisStation.x][thisStation.y] = "Station";

    }
    map[theNode.state.y][theNode.state.x] = map[theNode.state.y][theNode.state.x] + "|| CoastGuard " + "("
            + theNode.state.carriedPassengers + ")";

    for (int i = 0; i < map.length; i++) {
        System.out.println("");
        for (int j = 0; j < map[i].length; j++) {
            System.out.print(map[i][j] + ",");
        }

    }

}
```

Visualize takes in the node and makes a 2D array with the map size, and it loops over the ships and stations and writes their data within their respective locations within the grid, it also writes the coast guard's location in its place.

8.

9.

10.

11.

12. nearest_Human_ship

This function checks for the nearest ship that has humans on it by taking a certain position and calculating the distance between the Given position and every existing ship and returning the ship location that has the smallest distance .

```
public static int[] nearest_Human_ship(int x, int y, node thisnode) {

    int[] shipLocation = { -1, -1 };
    int min = 225;
    for (int i = 0; i < thisnode.state.ships.size(); i++) {
        if (thisnode.state.ships.get(i).are_there_people_here()) {
            int shipDistance = distance_to_target(x, y, thisnode.state.ships.get(i).x,
                    thisnode.state.ships.get(i).y);
            if (shipDistance < min) {
                min = shipDistance;
                shipLocation[0] = thisnode.state.ships.get(i).x;
                shipLocation[1] = thisnode.state.ships.get(i).y;
            }
        }
    }

    return shipLocation;

}
```

13. Nearest_station

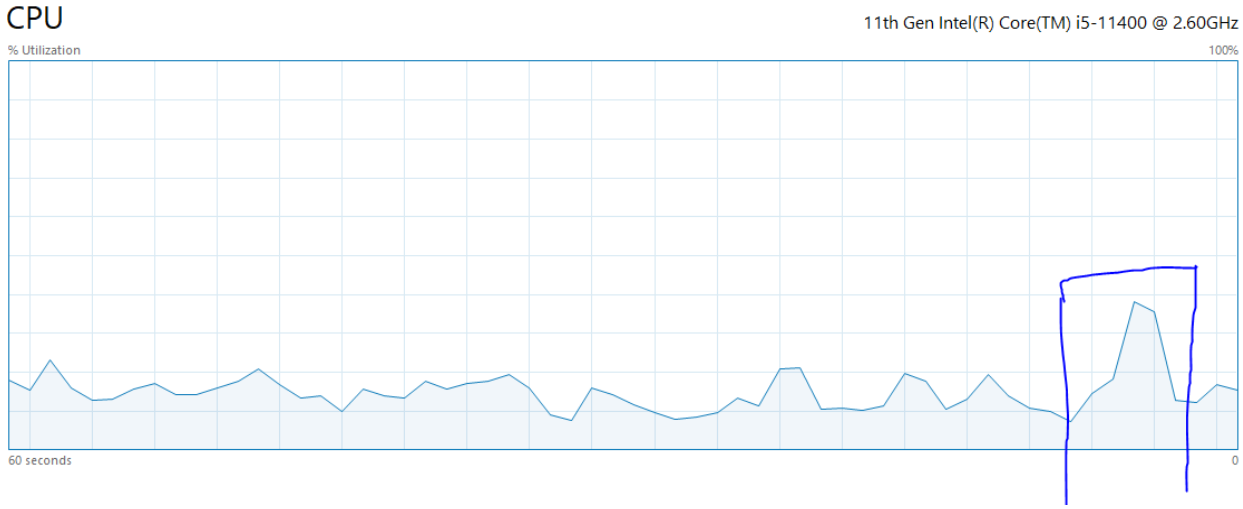
This function checks for the nearest station by taking the position of the coast guard and calculates the distance between the coast guard and every existing station and returns the station location that has the smallest distance.

```
public static int[] nearest_station(int x, int y, node thisnode) {

    int[] stationLocation = { -1, -1 };
    int min = 225;
    for (int i = 0; i < stationslist.size(); i++) {

        int shipDistance = distance_to_target(x, y, stationslist.get(i).x, stationslist.get(i).y);
        if (shipDistance < min) {
            min = shipDistance;
            stationLocation[0] = stationslist.get(i).x;
            stationLocation[1] = stationslist.get(i).y;

        }
    }

    return stationLocation;

}
```

Search Algorithms:

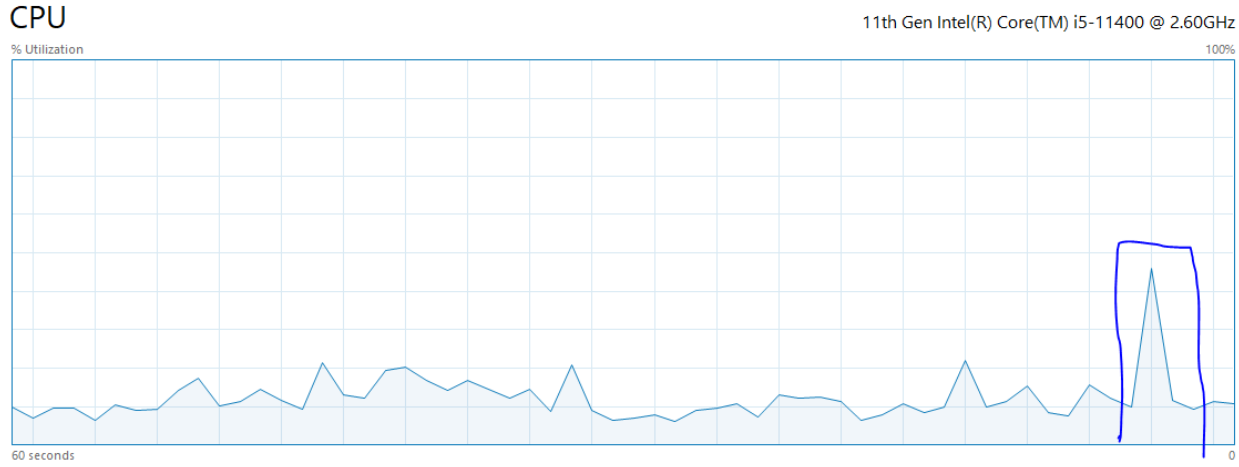
BFS:

In BFS, we expand every possible action as a node, since it's uninformed, and enqueue it at the end of the Queue. And dequeue the first element according to the general search algorithm, we also handled repeated cases using a hashmap, so it was considerably fast.



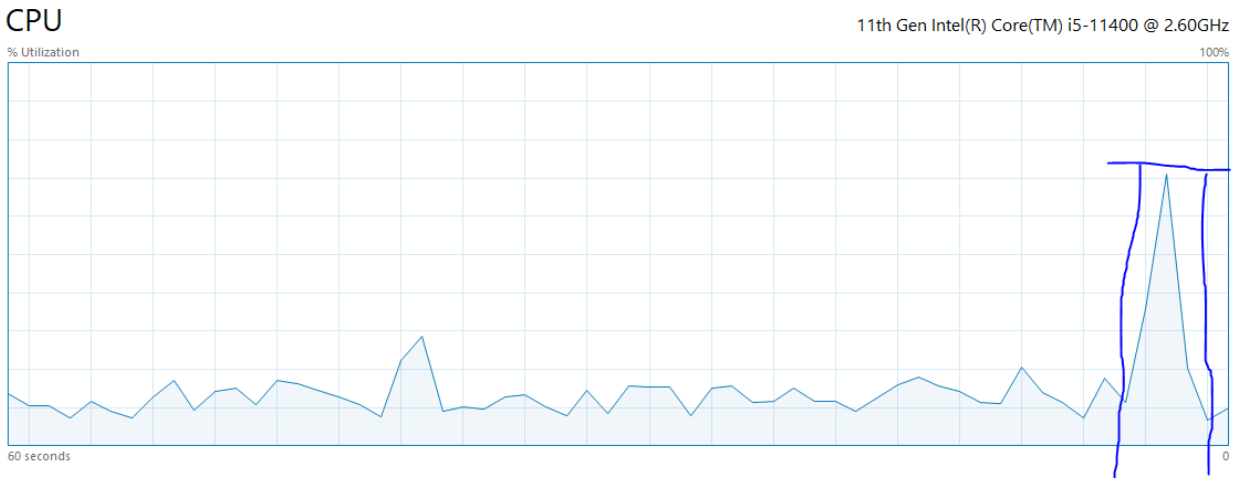
DFS:

In DFS, we expand every possible action as a node, since it's uninformed, and enqueue it at the first of the Queue. And dequeue the first element according to the general search algorithm. It runs faster than BFS.



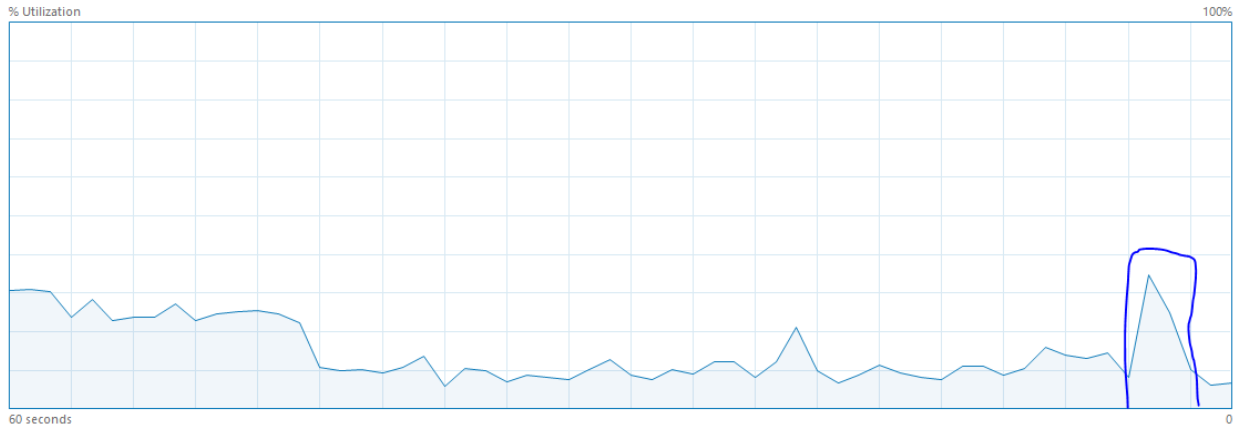
IDS:

In IDS, we created a global variable depth, and loop on the general search algorithm, and not expand nodes that are beyond the global variable depth, then increment the global variable until it finds a solution. It runs worse than DFS on average



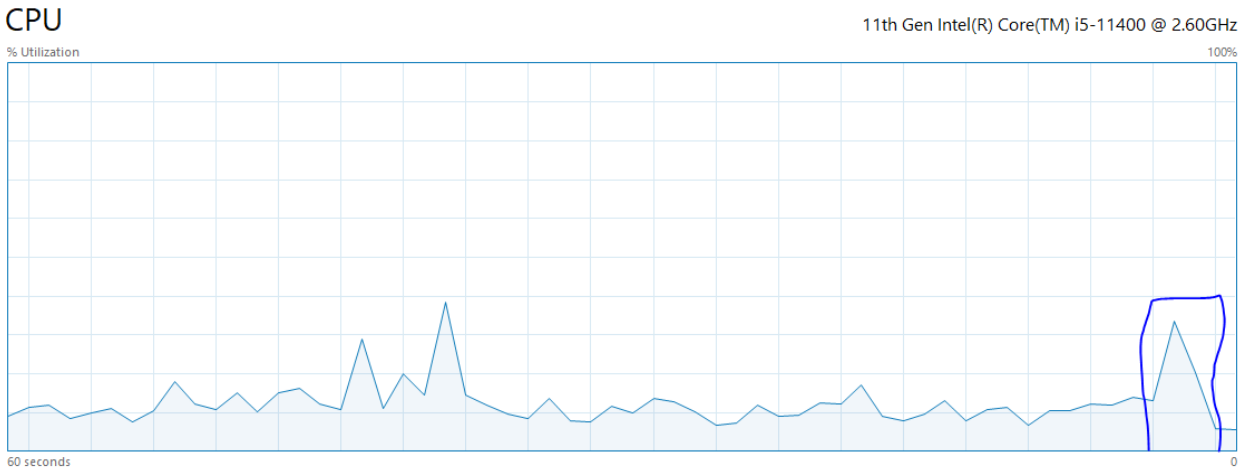
Greedy 1:

In Greedy 1, we expand every possible node, and enqueue them in a priority queue, according to Heuristic function 1 (introduced later).



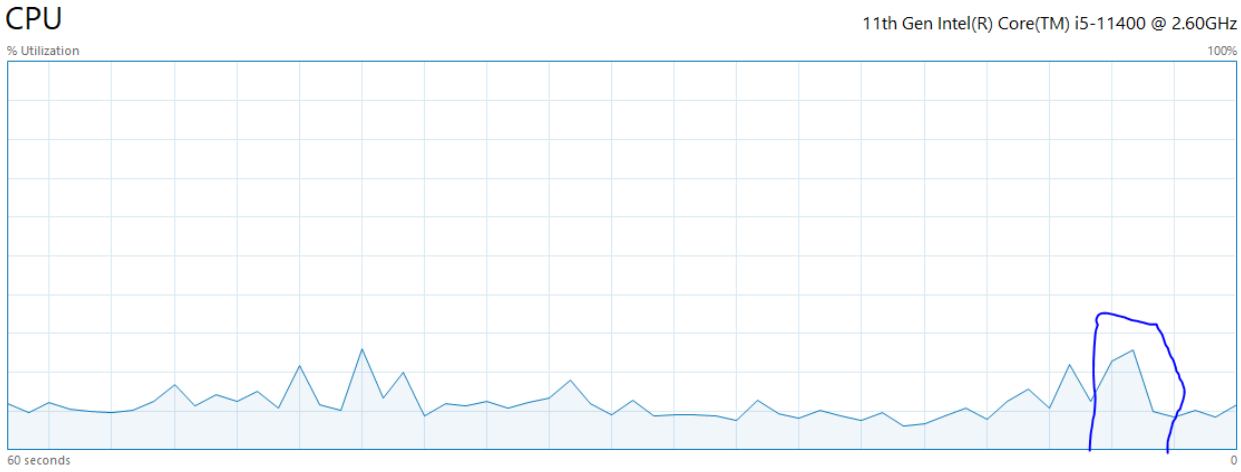
Greedy 2

In Greedy 2, we expand every possible node, and enqueue them in a priority queue, according to Heuristic function 2 (introduced later).



Astar1:

In Astar 1, we expand every possible node, and enqueue them in a priority queue, according to (Heuristic function 1) * comparator of pathCost.

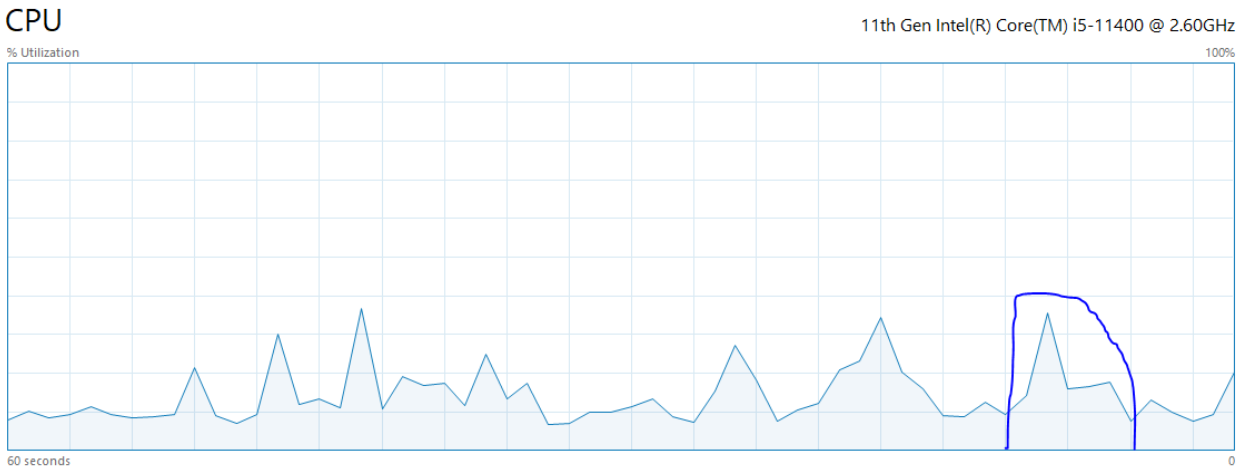


PathCost -> {deaths, lostboxes}

```
class UCScomparable implements Comparator<node>{
    public int compare(node node1, node node2) {

    if(node1.pathCost[0]> node2.pathCost[0]) {
        if(node1.pathCost[1] > node2.pathCost[1]) {
            return 1;
        }
        if(node1.pathCost[1] == node2.pathCost[1]) {
            return 1;
        }
        if(node1.pathCost[1] < node2.pathCost[1]) {
            return -1;
        }
    }
    else if (node1.pathCost[0]< node2.pathCost[0]) {
        if(node1.pathCost[1] > node2.pathCost[1]) {
            return -1;
        }
        if(node1.pathCost[1] == node2.pathCost[1]) {
            return -1;
        }
        if(node1.pathCost[1] < node2.pathCost[1]) {
            return -1;
        }
    }
    else if (node1.pathCost[0] == node2.pathCost[0]) {
        if(node1.pathCost[1] > node2.pathCost[1]) {
            return 1;
        }
        if(node1.pathCost[1] == node2.pathCost[1]) {
            return 0;
        }
        if(node1.pathCost[1] < node2.pathCost[1]) {
            return -1;
        }
    }

```

Astar2:

In Astar 2, we expand every possible node, and enqueue them in a priority queue, according to (Heuristic function 2) * comparator of pathCost.



Heuristics

1. priorityHumanDecider

For this heuristic first it checks if there are humans on a certain node or not if not then it checks if the coast guard reached max capacity if not it starts to search for the nearest ship that contain humans on it by calculating the distance between coast guard and other ships and returning the nearest distance to it but if the coast guard reached its max capacity it will check for the nearest station by also calculating the distance between the coast guard and every existing station and returning the nearest distance to an existing station. And thus it will return either the distance between the coast guard and the nearest station/ship and then we will use that as the cost for the function. Note: dropping off/picking up when near the ship takes highest priority (highest priority is the lowest value)

```
public static int priorityHumanDecider(node thisnode) {

    if (!are_there_humans(thisnode)) {

        if (!isFull(thisnode)) {
            int[] nearesthumanship = nearest_Human_ship(thisnode.state.x, thisnode.state.y, thisnode);
            int distance = distance_to_target(thisnode.state.x, thisnode.state.y, nearesthumanship[0],
                    nearesthumanship[1]);

            if (distance == 0 && thisnode.operator == operator.Pickup) {
                distance = -1;
            }
            return distance;
        } else {
            int[] nearestestation = nearest_station(thisnode.state.x, thisnode.state.y, thisnode);
            int distance = distance_to_target(thisnode.state.x, thisnode.state.y, nearestestation[0],
                    nearestestation[1]);
            if (distance == 0 && thisnode.operator == operator.Drop) {
                distance = -1;
            }
            return distance;
        }
    }
    return 0;
}
```

2. priorityBlackBoxDecider

For this one we check if there is a black box on a current node or not if not we start to search for the nearest ship that contain a black box by calculating the distance between coast guard on a current node and other existing black boxes And return the nearest distance to it

```
public int priorityBlackBoxDecider(node thisnode) {

    if (are_there_BlackBoxes(thisnode)) {
        int[] nearestBoxShip = nearest_Blackbox(thisnode.state.x, thisnode.state.y, thisnode);

        int distance = distance_to_target(thisnode.state.x, thisnode.state.y, nearestBoxShip[0], nearestBoxShip[1]);

        if (distance == 0 && thisnode.operator == operator.Retrieve) {
            distance = -1;
        }
        return distance;
    } else
        return 0;
}
```

Code Failures:

Our Code has some failures that do not lead to a goal state, which to our understanding has to do with the updating of the time step, which updates the states of every node, which would force the algorithm to end early or later.

References:

https://www.geeksforgeeks.org/comparator-interface-java/