**AI Project Report**

**A brief discussion of the problem**

On a grid of fixed width and height, sinking ships (containing passengers) are scattered randomly. For every time step, one passenger dies in each ship. If all passengers are dead, a black box becomes retrievable, but only within 20 time steps. The agent is a coast guard boat whose aim is to reach a state where there are no more passengers to rescue (by dropping them in stations that are also randomly scattered) and no more black boxes to retrieve. Optimally, the coast guard should do it in a way such that maximum number of passengers are rescued and maximum number of black boxes are retrieved. The agent can move up, down, left, right, can pickup passengers, can drop them in a station and can retrieve a black box. Each action takes one time step. In one time step, a passenger sinks in every ship.

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

A search tree node is a class that has attributes including:

1-Reference to the parent node.

2-The sequence of actions from the root node (that has initial state of the problem) to reach this node.

3-The path cost from the root: Tuple<The number of passengers who died, black boxes which became non-retrievable since the initial state of the root>.

4-State: A reference to the state object of the node where a state object is described by things that define a state:

1. Location of the coast guard
2. Map from the coordinates of a ship to the number of passengers currently in this ship
3. Map from the coordinates of a ship to the counter of the black box of the ship (initially zero and increments to 20).
4. Number of passengers currently on the coast guard.
5. Number of deaths and number of retrieved black boxes till now.
6. A function that performs a time step from the previous state by modifying the state with the events that happen once a time step happens (ex: decrementing number of passengers on each ship by 1).

5-Two functions that compute heuristic1 and heuristic2 of this node.

**A discussion of your implementation of the search problem ADT and A discussion of how you implemented the various search algorithms.**

A generic search problem is a class that has the following search functions. All the search functions are implemented as discussed in the lectures.

1. They begin with a queue that only has the root node state.
2. The nodes are enqueued based on an enqueuing method.
3. While the queues still has nodes, we dequeue these nodes based on priority of the queuing function.
4. If the dequeued node passes the goal test we return its sequence of actions, deaths, number of expanded nodes. Else we expand the node by trying to perform possible actions. Performing actions outputs children nodes which are also enqueued.

1-solveBreadthFirstSearch: where its queuing function depends on the depth. The lower the depth, the quicker a node is expanded.

2-solveDepthFirstSearch: where its queuing function depends on the depth. The higher the depth, the quicker a node is expanded.

3-solvedIterativeDeepeningSearch: where its queuing function depends on the depth. The higher the depth, the quicker a node is expanded to some limit l. If l limit is reached, l is incremented, queue is emptied and search is repeated with limit (l+1).

4-solveGreedySearch: where its queuing function depends on evaluating the first or second heuristic function that is implemented in class SearchTreeNode. The lower the heuristic result of the node, the quicker it is expanded.

5-solveAStarSearch: where its queuing function depends on evaluating the first or second heuristic function that is implemented in class SearchTreeNode plus the path cost of the node from the root. The lower the cost result of the node, the quicker it is expanded.

6-expandNode: an abstract method to expand the node by applying different actions. This method is implemented (has a body) in the CoastGuard child class. More details on this when discussing the CoastGuard class.

7-goalTest: an abstract method to check whether a node n has a state that passes the goal test. This method is implemented (has a body) in the CoastGuard child class. More details on this when discussing the CoastGuard class.

8-pathCost: an abstract method to evaluate the cost of reaching node n from the root. This method is implemented (has a body) in the CoastGuard child class. More details on this when discussing the CoastGuard class.

**A discussion of your implementation of the CoastGuard problem and A description of the main functions you implemented.**

The problem has:

-Operators: Move left, right, up, down, pickup, retrieve, drop

-Initial state: The initial grid with the ships’ locations and number of passengers, coast guard location

-State space: the states that are reachable from the initial state (Root of the tree) by any sequence of actions.

-Goal Test: Check if there are no more passengers, no more black boxes to retrieve on any ship and no more passengers on the coast guard.

-Path Cost: of a state is a tuple representing <the number of passengers who died from the root till this state, the number of black boxes which became non-retrievable from the root till this node>

The class CoastGuard is a child of the GenericSearchProblem class and has the following functions:

**1-GenGrid: generates a string of grid in the same format that is discussed in the project description.**

2-solve: takes a grid string, a strategy (type of search), visualize Boolean of whether the grid should be visualized during the search. This returns a string of “sequence of action;deaths;retrieved; number of expanded nodes”.

3-createGridFromString: A helper method that takes a grid string and creates a grid object using the Grid class that has the following attributes:

1. Width, Height
2. Max number of passengers on coast guard
3. A list of all coordinates that have stations
4. Initial coordinates of the coast guard

4-expandNode: this takes a parent node and outputs an arraylist of children nodes that result from performing possible actions. To avoid redundancy, we check whether each action is possible to perform before we actually output the child node resulting from performing it. For example, we check whether the coastguard is in a ship location, ship has passengers and coast guard has capacity, else the action is not performable.

5-Helper functions: canPickup, canMove, canDrop, etc. that check whether each action is performable as discussed.

6-Helper functions: move, pickup, retrieve, drop, where each one of them computes the effect of performing the action in the new state (for example: if action is move right, then we increment its x location by 1. If it is drop, number of passengers on the coast guard becomes zero in the new state.) and calls the perfromATimeStep in the State class to simulate what happens when a time step happens (ex. Number of passengers decrements by 1 in each ship).

7-goalTest: checks:

1. That there are no coordinates that have any ships.
2. That there are no more black boxes that are retrievable.
3. The coast guard has no passengers that need to be dropped.

8-pathCost: get path cost of reaching a specific node n, that is, tuple <the number of passengers who died, black boxes which became non-retrievable since the initial state of the root>.

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

1-First Heuristic Function:

The heuristic checks if the coast guard still has some capacity for picking up more passengers:

If true, then the cg will be searching for a ship. Therefore, the Mannhattan distance to the nearest ship is calculated. From this distance, the sum of deaths that will happen to this nearest ship is calculated and this is considered to be the heuristic.

If false, then the cg will be searching for a station. Therefore, the Mannhattan distance to the nearest station is calculated. From this distance, the sum of deaths that will happen to this nearest station is calculated and this is considered to be the heuristic.

The intuition behind this admissible heuristic is that reaching the nearest ship or station in each case represents a lower bound on the steps that the cg must do to lower the cost.

2-second heuristic function:

For the second heuristic we consider two values as a pair (Deaths to reach goal state, Number of black boxes to reach goal state)

For the number of black boxes we take black boxes in the grid and evaluate their values by this equation (20-(counter at the time I reach the box)) which will give negative value if the counter at reaching the black box reached 20 or exceeds it and using this we can calculate number of black boxes that will be damages before reaching them.

For the number of deaths we considered 2 cases, Either the coast guard ship has a free capacity to take more passengers or it has no more free capacity and needs to go drop passengers at nearest station

For the first condition (If the coast guard has free capacity), First, we will check that if no more people exist on the grid in the ships, The heuristic will be equal 0.

If there still exist people on ships the heuristic will calculate the distance from the coast guard to every ship on the grid and then to the nearest station to this ship, and then using this distance calculated it will calculate the number of people that will die if the coast guard rescued this ship and store them in a hashmap called (deathsToRescueShip).

We will calculate the number of people that can be rescued if we reach this ship by getting the minimum between the free capacity of the coast guard and the number of people on the ship.

Then we will evaluate every ship by calculating the number of people that will die if the coast guard decides to rescue this ship minus the number of people that the Coast guard will rescue in this ship minus.

We will take the ship with a minimum evaluation value which means less people dying and more people being rescued and then get the values of people dying for this ship and give it as a value for the heuristic function.

For the second case, which is if the coast guard has no capacity, We will simply compute the Manhattan distance between the coast guard and each station and compute the deaths that will occur if the coast guard reached this station. Then we get the station with the minimum number of deaths and make the heuristic value equals to the minimum number of deaths.

3-Third Heuristic Function:

Heuristic is the minimum of two situations:

Situation 1: The “ship with the max number of passengers” or “s” will be rescued. Therefore, we are sure that the number of passengers who will die is the manhattan distance from the coast guard till the ship.

Situation 2: the ship with the maximum number of passengers will sink, all passengers will sink. So cost will be the number of passengers on this ship.

Note that here we considered the ship “s” with the maximum number of passengers because it is the last one that completely sinks. Therefore it controls our path cost to the goal.

**A discussion of of the performance of the different algorithms implemented in terms of completeness, optimality, RAM usage, CPU utilization, and the number of expanded nodes :**

* **The following tests for Ram and Cpu have been made on Mac M1 pro with silicon ship**
* **We will conduct the tests on the same (grid5) from the test cases that has dimensions of 5 x 5 as an average size gird to apply all tests on**
* **I have called java functions that measures the Ram and Cpu time of each processand will include screenshots of the output of each one**

**Packages used are :**

1. **com.sun.management.OperatingSystemMXBean;**
2. **import java.lang.management.ManagementFactory;**

**To compute both those functions are implemented inside compute() function in Generic search problem java class and then called if the goal state is reached in each solve method:**

1. **OperatingSystemMXBean osBean = ManagementFactory.getPlatformMXBean (OperatingSystemMXBean.class);**
2. **Runtime runtime = Runtime.getRuntime();**
3. **System.out.println("Cpu Usage Time in milliseconds is " + osBean.getProcessCpuTime() / 1000000 );**
4. **long memory = runtime.totalMemory() - runtime.freeMemory();**
5. **System.out.println("Used memory in bytes: " + memory);**

**1)Breadth First Search:**

* Completeness: Complete as it has reached the goal state
* Optimality: Not optimal as it can lead to a lot of people dying as it is uninformed search that does not put the cost function in consideration
* Ram Usage: 51832296
* CPU Run Time : 888 milliseconds
* Number of expanded Nodes: 36644

**2)Depth First Search:**

* Completeness: Complete as it has reached the goal state .
* Optimality: Not optimal as it can lead to a lot of people dying as it is uninformed search that does not put the cost function in consideration and it will reach a child node as fast as possible and consider it a goal state.
* Ram Usage: 4437792 Bytes
* CPU Run Time : 124 Milliseconds
* Number of expanded Nodes: 99

**3)Uniform Cost Search**

* Completeness: Complete as it has reached the goal state .
* Optimality: Optimal as it will always reach a goal state with the least path cost.
* Ram Usage: 2410528 Bytes
* CPU Run Time : 546 Milliseconds
* Number of expanded Nodes: 815

**3)Iterative Deepening search**

* Completeness: Complete as it has reached the goal state .
* Optimality: Not optimal as it can lead to a lot of people dying as it is uninformed search that does not put the cost function in consideration and it will reach a child node as fast as possible and consider it a goal state.
* Ram Usage: 61334632 Bytes
* CPU Run Time : 949 Milliseconds
* Number of expanded Nodes: 101609

**4)Greedy with first Heuristic**

* Completeness: Complete as it has reached the goal state .
* Optimality: Not optimal but it minimizes the cost to reach the goal state based on the heuristic.
* Ram Usage: 4650928 Bytes
* CPU Run Time : 305 Milliseconds
* Number of expanded Nodes: 190

**5)Greedy with the second Heuristic**

* Completeness: Complete as it has reached the goal state .
* Optimality: Not optimal but it minimizes the cost to reach the goal state .
* Ram Usage: 4492968 Bytes
* CPU Run Time : 312 Milliseconds
* Number of expanded Nodes: 86

**6)A\* with the first heuristic**

* Completeness: Complete as it has reached the goal state .
* Optimality: Optimal .
* Ram Usage: 5003024 Bytes
* CPU Run Time : 303 Milliseconds .
* Number of expanded Nodes: 571 .

**7)A\* with the second heuristic**

* Completeness: Complete as it has reached the goal state .
* Optimality: Optimal .
* Ram Usage: 5343712 Bytes
* CPU Run Time : 385 Milliseconds
* Number of expanded Nodes: 425
* **The Second heuristic is more complex than the first one so, number of expanded nodes is less and it uses more Ram and CPU since it requires more computations. In addition, the number of expanded nodes in both A\* heuristics is less than the number of expanded nodes in the UCS algorithm. This is because A\* is optimally efficient. Other algorithms may expand less nodes but they are not optimal.**
* **The number of expanded nodes in the Iterative Deepening Search is the most since it iterates on expanding the tree several times till reaching the goal.**
* **CPU run-time is relatively reduced in A\* compared with other algorithms. This is because A\* is more efficient in terms of expanding nodes as it only expands nodes that may lead to an optimal state in terms of cost and heuristic.**
* **RAM Usage in A\* and greedy is relatively high because the heuristics require the initialization of some useful data structure that help in the calculation of the heuristic****.**