# AI Project 1 Report

TEAM ,Abokammer M.,Khair O.,Mohsen M.

December 8, 2022

## 1 THE PROBLEM

This project is meant to solve a problem called the coast guard problem. this problem is briefly a situation where the world is a grid having damaged ships containing people, stations, and a single coast guard. The problem aims to save as many people as possible by carefully choosing the actions at each step, and also saving as many black boxes from the damaged ships as possible. Generally, it's a search problem for an optimal solution to a goal which is to save people & and black boxes and move them to the stations until there are no more to save. We can consider the coast guard itself as the agent in this world.

## 2 DESIGN

While executing project, We aimed at two goals. The first is to implement a quite generic framework so that after properly describing any search problem - following the guidelines of this framework- the framework can solve it. The second is to follow the functional style as much as we can - and as java would hopefully allow us- so that we can easily translate the algorithms described formally in the book to code.

### 2.1 Problem

As we mentioned, we intended to convert the formal definition to code. So, as the problem is formally defined as a five-tuple of (operations, initial state, state space, goal test, and path cost), we designed the **Problem** abstract class to have the needed fields from that tuple as depicted here in the UML design of the problem Figure 1, where *getPossibleOperations* method returns the operation as function that applies the operation on a node and returns the new node.

### 2.2 Node

Following the definition of the node, we can see that it's defined as a five-tuple (state, parent, operator, depth, path cost). We constructed the class to have the same as the five-tuple but replaced the operator with an action string to be able to know which action results in this node, and also we added the *compareCost* attribute to be able to compare between nodes when needed. we separated that from the *pathCost* as it's not always the case that we compare with path cost only. The class description is shown in Figure 1.

## 2.3 State

Modeling the state was the most challenging part; it varies a lot from one problem to another. So in order to model it cleanly - as far as java allows - we found that injecting it as a dependency to the classes that use it was the least problematic approach, rather than using reflection or unsafe class casting. That justifies why we see that the $Problem$, $Node$, $GeneralSearch$ are Generic Classes in - $T$ represents the state Class in Figure 1. This allows us to move the business logic - problem-specific logic- to its rightful place and decouples the search class from the problem. the search class only takes some operations as functions and applies them to nodes and puts these nodes in a queue with some enqueuing policy. with that, we somewhat achieved a general framework to solve search problems.

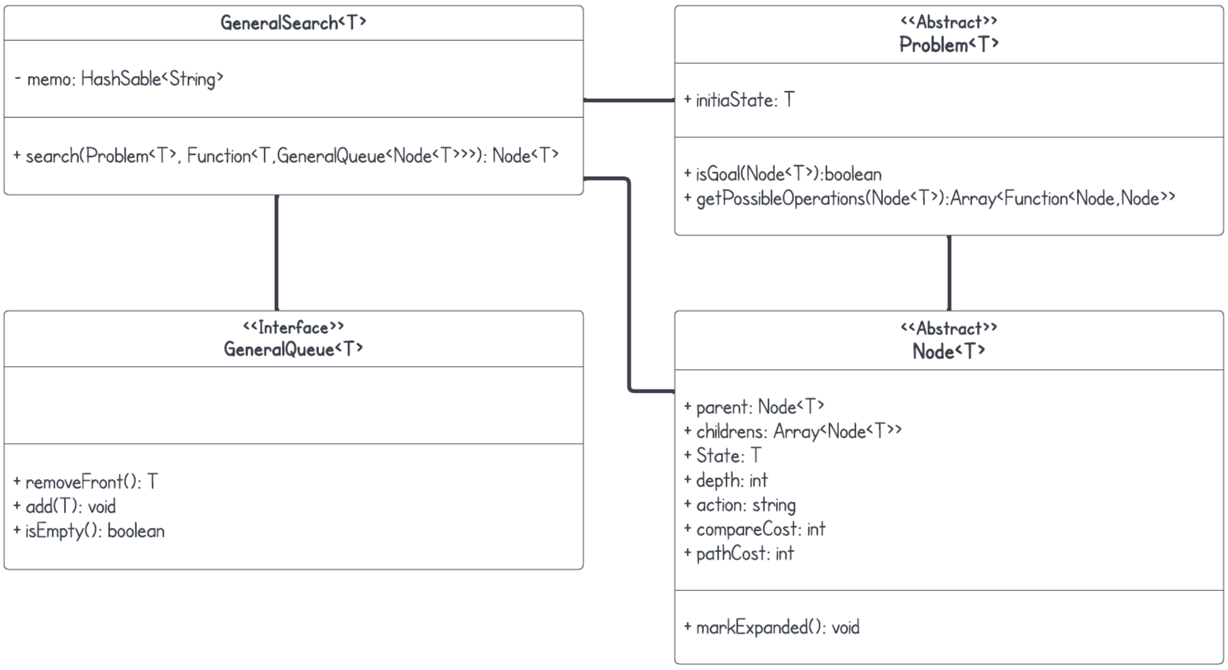

Figure 1 – UML diagram for the ADTs

## 2.4 General Search

With all of that described, the implementation of the general search was very easy, it was a direct translation from the pseudocode. The search function takes a problem and a $makeQueue$ function to benefit from the strategy pattern and decouples the search from the enqueuing policy. We uses $makeQueue$ instead of $QuinFunc$ as in the original pseudocode; because java is an OOP language, so the enqueuing policy is specified in the creation of the queue. We also added a $GeneralQueue$ interface, to be able to use the datastructures from the standard library in a consistent and unified mannar. The simplified general search is shown in this code snippet 2.4.

```java
public Node<T> search(Problem<T> p, Function<T, GenericQueue<Node<T>>>
    makeQ) {
    GenericQueue<Node<T>> queue = makeQ.apply(p.initialState);

    while (true) {
        if (queue.isEmpty())
            return null;
        Node<T> node = queue.removeFront();
        node.mark_expanded();

        if (p.isGoal(node))
            return node;

        for (Function<Node<T>, Node<T>> op : p.getPossibleOperations(node))
             {
            Node<T> newNode = op.apply(node);
            newNode.parent = node;
            queue.add(newNode);
        }
    }
}
```

## 2.5 CoastGuard Problem

After laying the abstract classes and interfaces, the only thing left was to implement the *CoastGuard* Problem itself, the *CoastGuard* Problem, and *CoastGuardState*. The *CoastGuardState* class represents the problem state containing the following:

- $gridW : int, gridH : int$ as a pair of integers to represent the dimension of the grid.
- $stations : Set < Pair < int, int >>$ represents the locations of the stations in the grid, both of this attribute and the $gridW, gridH$ are static attributes because they are not meant to be changed from one state to the other.
- $ships : Map < Pair < int, int >, Ship >$ it's a map between the ship positions to an object that holds the ship state of number of alive people and the life of the black box.
- $pos : Pair < int, int >$ represents the position of the coast guard itself.
- $retrievedBoxes, deadPassengers, savedPassengers$ represents the number of retrieved boxes so far, and the number of dead and saved passengers so far.

The *CoastGuard* problem class extends the $Problem < CoastGuardState >$ abstract class, and only defines the operations that can be done, and the heuristic functions - as well as some helper functions to serialize and visualize the result. We modeled the heuristic functions as functions that takes a node and returns the estimation as an integer value. On the other hand, we modeled the operation functions as functions that take a node and returns a new node after applying the specified operation. The *getPossibleOperations* is an important function as it specifies when we can perform an action. As a result of the generic design of the *GeneralSearch* we can now solve the coast guard problem without any steps other than specifies the queuing function.

# 3 SEARCH ALGORITHMS

To fully cover all the search algorithms, the *CoastGuard* problem class has two attributes to control which cost contributes to comparing different nodes, the $usePathCost : boolean$, $heuristic : Function < Node, Integer >$. Other than this, the search algorithms can be implemented only be choosing the right data structure.

#### 3.0.0.1 BFS

BFS can be achieved by using Queue as follows 3.0.0.1; here *GQueue* is wrapper around the standard library queue but implements the *GeneralQueue* interface.

```
generalSearch.search(
    problem,
    (state) -> {
        GenericQueue q = new GQueue();
        Node node = new Node(state);
        q.add(node);
        return q;
    }
);
```

#### 3.0.0.2 DFS

Same as BFS the only difference is just we will use stack this time instead of queue. We provided a wrapper for the standard library stack called *GStack*.

#### 3.0.0.3 IDS

To support the IDS algorithm we added an extra attribute for the *GeneralSearch* class 2.4 called $MaxDepth = -1$ by default it contains $-1$ indicating that there is no limit for the depth, otherwise the search procedure will not go further than the specified depth. With this, we can use the same *makeQ* function of the DFS and loop while increasing the depth as long as there are no solution found.

#### 3.0.0.4 UCS

Same as the BFS with replacing the queue with priority queue - *GPriorityQueue*. Also, we need to make sure that *usePathCost* attribute of the *CoastGuard* class is set to *true* and the *heuristic* attribute is set to *null* to compare nodes by the path cost only.

#### 3.0.0.5 Greedy Search

Same as the UCS with difference of the *CoastGuard* attribute configuration. We need to set the *usePathCost* to $false$ and to set *heuristic* to the reference of the desired heuristic function to compare nodes based on the heuristic function only.

#### 3.0.0.6 A* Search

Same as the UCS, but in this case we set the *usePathCost* attribute to *true* and the *heursitic* to the desired function to combine both costs and perform the A* Search.

## 4 HEURISTIC FUNCTIONS

We followed the suggested technique to construct heuristic functions, which is to relax some constrains. Let's first talk about our cost functions. We defined our cost function as a weighted sum of the loss of the dead passengers and the lost black box, so at each step the cost of the action taken is the number of dead passengers and lost black boxes in this step.

### 4.1 First Heuristic Function

The first heuristic named $heuristicFunc1$ in the code. It relaxes the movement and capacity constrains by estimating the cost to the nearest goal state as the number of the remaining ships. It's admissible as to reach a goal state there must not be any passengers that we can save or black boxes we can retrieve. So estimating with the number of ships can never over estimate the actual cost.

### 4.2 Second Heuristic Function

The second heuristic named $heuristicFunc2$. It relaxes the number of ships and the capacity by estimating the remaining cost as if there is only one ship left and it's the closest to the Coast Guard. It's admissible because it only consider one ship and if there is only one ship relaxing the capacity of the coast guard makes sure that the estimated cost will never overshoot, and if there are more than one it's guaranteed to be admissible because the actual cost will contain the lose from the other ships.

## 5 COMPARISON

### 5.1 Completeness & Optimality

We can summarize the Completeness and Optimality of the different algorithms in the following table.

| | BFS | DFS | IDS | UCS | GR | AS |
|---|---|---|---|---|---|---|
| Completeness | C | $C^c$ | C | C | NC | C |
| Optimality | NO | NO | NO | $O^c$ | NO | O |

The condition for the DFS to be Complete is to have no repeated states, meaning it's guaranteed to reach a goal state. Note also BFS and IDS could be optimal if the cost of all the operations is constant at least per level but that's is not meet in this problem. The condition for the UCS to be optimal is to have none decreasing cost function (there is no operations with negative costs) and that's satisfied in this problem.

### 5.2 Notes for CPU & Memory

The CPU & Memory metrics is sampled with profiler called $JProfiler$, but the sampling in not 100% accurate, also it's sampled from our personal computer so there are some uncertainty from the OS. On Last Note, the profiling results has more dimensions than the theoretical aspects of the algorithms. For the following plots we chose some of the provided unit tests (like testx9) to compare between the different search algorithms.

## 5.3 CPU Utilization

The following plot 2 shows the avg CPU utilization in terms of (Process and System time). The difference between the process time and the system time is what the program does, meaning that is it doing OS-related stuff like syscalls, or is it doing common process operations? We can see that the BFS has the most CPU utilization and that can be justified by the heavy usage of memory with a dynamic data structure which necessary requires more syscalls to allocate more memory, also the java GC (Garbage collector) will have more work to perform which is reflected in the CPU utilization. The same thing can be said in the case of IDS as it repeatedly allocates memory for new objects (the impact of that will be clear in the RAM usage plot). Other than that, we can see that the other algorithms are taking reasonable percentages of the CPU. On last note here is that we can see that the DFS CPU utilization is very small to the extent that it can't be captured by the profiler and that's because it's a simple algorithm, it takes a path until it reaches a goal.

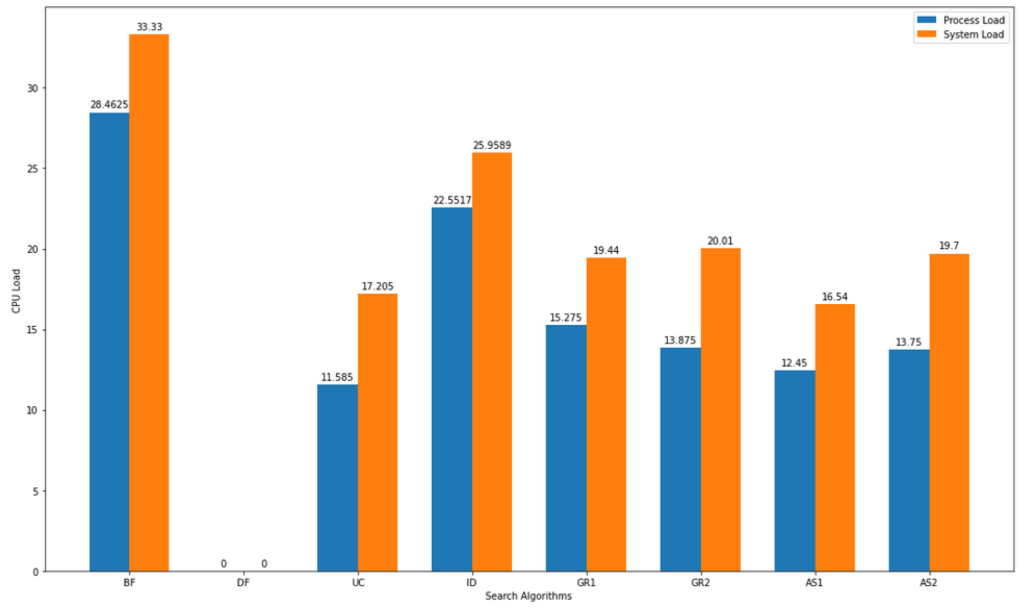

Figure 2 – CPU Load

## 5.4 RAM Usage

The following plot 3 is showing the RAM utilization of the different algorithms implemented in java. We can see that the optimal algorithms and the greedy ones has low memory footprint as they are expected to expand small numbers of search nodes. The shocking thing about this graph is the memory usage of the IDS algorithm compared to the BFS algorithm, because on the contrary of what the advantage of the IDS over the BFS, we see that it actually consumes more memory! But that behaviour can be explained when we take the Java language runtime into account. As we know java is a garbage collected language, which allocates memory for objects and tries to free this memory when its unused, but to overcome the performance issues it runs with some heuristics. The behaviour of the GC is out of our context in this project, but we can conclude that GC allocates more memory than needed to avoid out of memory faults, and it does not collect all the garbage memory at the moment it's being unused (inaccessible). Also the pattern of memory allocations affects the GC collection behaviour. With that said, we can see that plot is somewhat expected as the IDS algorithm allocates and frees memory so often in an attempts to not hold a lot of memory as the DFS does, which results in that huge memory footprint.

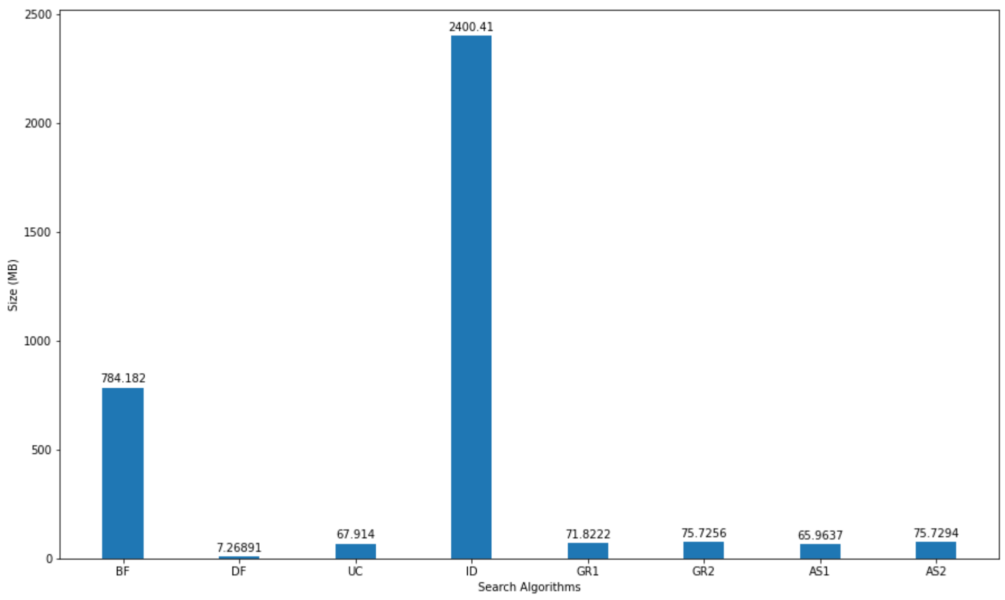

Figure 3 – Memory Usage

## 5.5 Expanded Nodes

The following plot 4 shows the comparison between the different algorithms. We can see that the DFS has the least expanded nodes that's because it takes the first possible action and continue doing it until it reaches the goal without the need for explore other options. We can also see that the A* Algorithms expanded exactly the same nodes as the UCS, which meets its constrain of not expanding more than any other optimal algorithm.

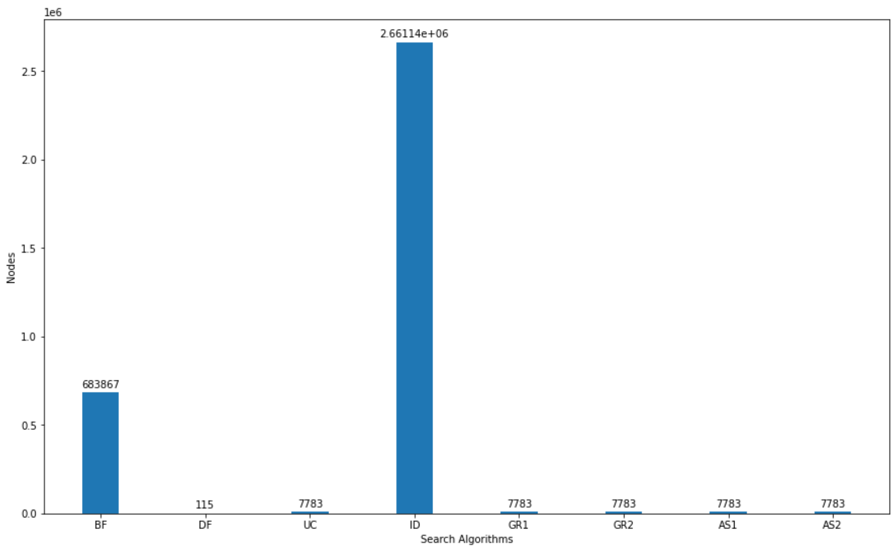

Figure 4 – Expanded Nodes