AI Project 1 Report

Rowan Amgad and Maryam ElOraby
Team Name: Rational Agents

9 December 2022

1 Problem

A coast guard is in charge of a rescue boat that goes into the sea to rescue other sinking ships. When rescuing a ship, they need to rescue any living people on it and to retrieve its black box after there are no more passengers thereon to rescue. If a ship sinks completely, it becomes a wreck and they still have to retrieve the black box before it is damaged. Each ship loses one passenger every time step. Additionally, each black box incurs an additional damage point every time step once the ship becomes a wreck. One time step is counted every time an action is performed. **Goal is reached when there are no living passengers who are not rescued, there are no undamaged boxes which have not been retrieved, and the rescue boat is not carrying any passengers**. They would also like to rescue as many people as possible and retrieve as many black boxes as possible. This could be thought of as a search problem defined as follows:

1. **Operators:** `Up, Down, Left, Right, Pick-up, Drop, Retrieve`
2. **Initial State**: Coast guard, ships and stations are at random locations within the grid boundaries. Each ship has a random number of passengers p onboard, where $0 < p \leq 100$. There are no wrecks, and no two items are in the same cell.
3. **State Space:** Each state represents grid dimensions, maximum number of passengers the coast guard rescue boat can carry, current coast guard location, current locations of all stations, current locations of all ships and number of remaining passengers on each ship, current locations of all wrecks and the counter number of each black box, current remaining capacity in the coast guard rescue boat, remaining passengers on the grid, remaining ships on the grid, remaining wrecks on the grid, dead passengers so far, retrieved boxes so far and the lost boxes so far.
   It is represented as follows: `griddims; maxCapacity; location of coast guard; locations of stations; locations of ships and # of passengers on each ship; locations of wrecks and the current damage; remaining`

`cg capacity; # remaining ships; # remaining boxes; # dead passengers ; # retrieved boxes; # lost boxes.`

4. **Goal Test:** No living passengers on grid. No passengers on rescue boat. No unretrieved undamaged boxes.

5. **Path Cost:** $< deaths, lostboxes >$

2 Search Algorithms

2.1 Breadth-First Search

The Breadth-First Search (BFS) algorithm starts at the root node and explores the search tree level by level. The algorithm is guaranteed to find a solution given that it exists and the number of operators is finite as under this assumption (which in most use cases is valid), the algorithm will not get stuck. For our implementation, we used a queue that dequeues the tree's nodes level by level starting from the root node. The root node is inserted into the queue and poll operations are performed until the queue is empty or a node that satisfies the goal condition is reached.

2.2 Depth-First Search

The Depth-First Search (DFS) algorithm starts at the root node and explores the search tree branch by branch. In general, this algorithm is not guaranteed to find a solution if one of the search tree's branches or more are infinite, which is the case in many practical problems. However in our case it is complete and guaranteed to find a solution if exists because repeated states are not expanded and hence the search tree becomes finite. For our implementation, we used a stack that pops the tree's nodes level by level within a specific branch starting from the root node and then goes back to traverse the unvisited branches. The root node is inserted into the stack and pop operations are performed until the stack is empty or a node that satisfies the goal condition is reached.

2.3 Iterative Deepening Search

The Iterative Deepening Search (IDS) is a modification to the Depth-First Search that aims to solve the problem that DFS is not guaranteed to find a solution while maintaining its low space complexity. IDS works by iteratively performing DFS up to a limited maximum depth and increasing this max after each iteration. In our implementation, a stack is used to traverse the tree's branches within a loop that increments the maximum depth with each iteration. Within each iteration, The root node is inserted into the stack and pop operations are performed until the stack is empty or a node that satisfies the goal condition is reached.

2.4 Greedy Search

The Greedy Search utilizes a heuristic function which estimates the cost from a given node to the nearest goal node. Such an estimation can not be exact without exploring the entire search space which would defeat the purpose of the search. Instead, estimation is calculated from the current node to a near node to the goal node. In our implementation, nodes are enqueued into a priority queue according to the result of the heuristic function. The root node is inserted into the queue and poll operations are performed until the queue is empty or a node that satisfies the goal condition is reached.

2.5 A* Search

The A* Search is the second optimal algorithm that we implemented. A* also takes advantage of the heuristic function utilized by Greedy Search, however, what makes A* optimal is that it adds the path cost to the result of the heuristic function and uses the result of this addition to enqueue nodes in a priority queue. The root node is inserted into the queue and poll operations are performed until the queue is empty or a node that satisfies the goal condition is reached.

3 Node ADT

Node implements comparable class. Node also contains the attributes:

- Current State
- Node's parent
- Action taken to reach this Node
- Depth of node
- Path cost $< deaths, lostboxes >$
- Final path cost: *path cost to be used to enqueue nodes. Differs according to the search strategy:*
  - UC: finalPathCost = "actual deathSoFar,actual lostBoxes"
  - GR1: "estimated deaths,0" and GR2: "0,estimated lost boxes"
  - AS1: "estimated deaths + actual deaths, actual lost boxes" and AS2: "actual deaths, estimated lost boxes + actual lost boxes"

  Functions specific to Node class:

  - `getAncestors()`: gets the path that leads to current node from Node. This is done through iteratively calling the Nodes' parents till the root node.
  - `compareTo(Node n)`: overrides `CompareTo()` method to compare two nodes based on their final path costs. The priority is given to the number of deaths which we aim to minimize.

4 Search Problem ADT

Search Problem class is an abstract class that contains 2 methods:

- `solveSearchProblem(String grid, String strategy)`: A method that takes as input a grid ina string form and a search strategy. This method shall be overriden by CoastGuard class and be used to solve the grid according to the input strategy.
- `isGoal(Node node)`: A method that takes as input an object of type "Node" and shall be overridden by CoastGuard class in order to check whether the input node is a goal node or not.

5 CoastGuard Problem Implementation

The CoastGuard problem is solved using the `solve(String grid, String strategy, boolean visualize)` method which calls the `solveSearchProblem(String grid, String strategy)` method that's overridden in the `CoastGuard` class that solves the CoastGuard problem by executing the following steps in order:

1. Create an initial state of the world using `createInitialState(String grid)` method.
2. Create a root node using the initial state
3. Call one of `BF()`,`DF()`, `ID()`, `GR1()`,`GR2()`, `AS1()` or `AS2()` in order to solve the search problem according to the input strategy.

In addition, if `visualize` value was `true`, the method `visualizeState(String currState)` is called in order to print the initial state of the world and its state after each action taken that lead to the goal node.

6 Main Functions

- `String genGrid()`: Randomly generates a grid where the dimensions of the grid, the locations of the coast guard, stations, ships, number of passengers for each ship and the coast guard capacity are all to be randomly generated. A minimum of 1 ship and 1 station are generated and there is no upper limit for the number of ships or stations generated. In addition, no 2 objects occupy the same cell.
- `isGoal(Node n)`: Checks if the input node n is a goal node or not by checking if there are no remaining alive unsaved passengers, no remaining black boxes to be retrieved and no remaining passengers on the coast guard's rescue boat.

- `visualizeState(String currState)`: Visualises the state of the world in a grid: Displays the coast guard in its location as `cgX` where x is the maximum capacity of the guard. When the guard is on a cell with another object, it is marked as `(c)`. It displays the ships in their locations as `XshY` where X is the current number of passengers on the ship and Y is the unique index of the ship (just to differentiate ships from each other). Displays the stations in their locations as `stY` where Y in this case is a unique index of the station. Finally, it displays the boxes/wrecks on the grid as `XwrY` where X is the current damage of the wreck and Y is the unique index of the box/wreck. An example of this is the following:

```
+---------+---------+---------+
|    *    |   st0   |    *    |
+---------+---------+---------+
|    *    |    *    |  cg97   |
+---------+---------+---------+
|    *    |    *    |    *    |
+---------+---------+---------+
|  5wr0   |    *    |  63sh0  |
----------+---------+---------+
```

- `createInitialState(String grid)`: Creates an initial state of the world by parsing the input grid, formulate an initial state consisting of all the grid's information in addition to the total number of passengers in the grid and other parameters that will be used to solve the search problem such as remaining coast guard rescue boat capacity, remaining alive passengers, remaining ships, remaining boxes to be retrieved, dead passengers so far and lost boxes so far.
- `generateNextState(String currState, Actions nextAction)`: Generates the next state of the world according to the action to be performed in order to reach that next state.
- `expandNode(Node n)`: Generates all possible children nodes of Node n without taking repeated states into consideration.

7 Heuristics

In this project, we implemented 2 heuristic functions which are `heuristic1(String currState)` and `heuristic2(String currState)`.

7.1 Heuristic 1

This heuristic is used to estimate the number of expected passengers to die before a goal state is reached. This estimation is calculated by executing the following steps in order:

1. Get the current coast guard rescue boat location, all the ships' locations and the number of remaining passengers on each ship in the current state.
2. Get the number of time steps needed to reach the nearest ship to the coast guard rescue boat according to the city block distance.
3. Calculate the number of passengers that will die on all ships if the coast guard rescue boat was to go the the nearest ship.

The heuristic is admissible because it calculates the number of passengers that will die if the coast guard rescue boat will go and rescue the nearest ship's passengers only and not take into consideration rescuing other ships' passengers as in order to rescue the rest of the passengers, more passengers will die in the process and hence the returned estimation will never be an over estimate of the actual remaining passengers to die before reaching a goal node. This heuristic is used in `GR1()` and `AS1()`.

7.2 Heuristic 2

This heuristic is used to estimate the number of expected boxes to be lost before a goal state is reached. This estimation is calculated by executing the following steps in order:

1. Get the current coast guard rescue boat location, all the wrecks' locations and the counter number on all black boxes on each wreck in the current state.
2. Get the number of time steps needed to reach the every wreck and retrieve the black box according to city block distance between the wreck and the current coast guard rescue boat location. If the box is to expire before the coast guard rescue boat is able to reach it and retrieve it, count it as a lost box.

The heuristic is admissible because it estimates the boxes that will be lost till a goal node is reached with respect to each wreck individually. In reality, the lost boxes are calculated according to the path of the coast guard rescue boat among all the wrecks, which means that more boxes than the estimated ones might be lost during the process and hence the returned estimation will never be an over estimate of the actual boxes that will be lost before reaching a goal node. This heuristic is used in `GR2()` and `AS2()`.

8 Input/Output Formats

8.1 Inputs

The inputs to `solve()` are:

1. *grid*: `M;N;C;cgX;cgY;I1X;I1Y;I2X;I2Y;..:IiX;IiY;S1X;S1Y;S1Passengers; S2X;S2Y;S2Passengers;..:SjX;SjY;SjPassengers;`
   where:
   - `M` and `N` represent the width and height of the grid respectively.
   - `C` is the maximum number of passengers the coast guard boat can carry at time.
   - `cgX` and `cgY` are the initial coordinates of the coast guard boat.
   - `IiX`; `IiY` are the `x` and `y` coordinates of the `ith` station.
   - `SjX`; `SjY` are the `x` and `y` coordinates of the `jth` ship.
   - `SjPassengers` is the initial number of passengers on board `jth` ship.
2. *strategy:*
   - BF for breadth-first search,
   - DF for depth-first search,
   - ID for iterative deepening search,
   - GRi for greedy search, with i distinguishing the two heuristics.
   - ASi for A search with i distinguishing the two heuristics.
3. *visualize* is a Boolean parameter, if set to *true* it displays a visualization of the path taken to the goal.

8.2 Output

The result is returned in the following format: `plan;deaths;retrieved;nodes` , where:

- `plan`: the sequence of steps and actions that lead to the goal.
- `deaths`: number of passengers who have died in the solution starting from the initial state to the found goal state.
- `retrieved`: number of black boxes successfully retrieved starting from the initial state to the found goal state.
- `nodes`: is the number of nodes chosen for expansion during the search.

9 Results and Discussions

Take for example the following grid input: `6,7;82;1,4;2,3;1,1,58,3,0,58,4,2,72;` visualized as:

```
+--------+---------+---------+---------+---------+---------+
|    *   |    *    |    *    |    *    |    *    |    *    |
+--------+---------+---------+---------+---------+---------+
|    *   |  58sh0  |    *    |    *    |  cg82   |    *    |
+--------+---------+---------+---------+---------+---------+
|    *   |    *    |    *    |   st0   |    *    |    *    |
+--------+---------+---------+---------+---------+---------+
|  58sh1 |    *    |    *    |    *    |    *    |    *    |
+--------+---------+---------+---------+---------+---------+
|    *   |    *    |  72sh2  |    *    |    *    |    *    |
+--------+---------+---------+---------+---------+---------+
|    *   |    *    |    *    |    *    |    *    |    *    |
+--------+---------+---------+---------+---------+---------+
|    *   |    *    |    *    |    *    |    *    |    *    |
---------+---------+---------+---------+---------+---------+
```

the initial state will be: `6,7;82;4,1;3,2;1,1,58,0,3,58,2,4,72;$;82;188;3;0;0;0;0`
The results for each strategy were as follows[1]:

1. **BF**:
   - Result: `DOWN,DOWN,DOWN,LEFT,LEFT,PICKUP,UP,UP,RIGHT,DROP,DOWN,LEFT, LEFT,LEFT,PICKUP,RETRIEVE,UP,UP,RIGHT,PICKUP,RETRIEVE,DOWN,RIGHT, RIGHT,DROP,UP;39;2;88968`
   - Maximum CPU Utilization: `35.0%`
   - Maximum Memory Utilization: `368.4 MB`
2. **DF**:
   - Result: `LEFT,LEFT,LEFT,PICKUP,RETRIEVE,LEFT,DOWN,RIGHT,RIGHT,RIGHT,DROP, LEFT,LEFT,LEFT,DOWN,PICKUP,RETRIEVE,RIGHT,RIGHT,RIGHT,RIGHT,RIGHT,DOWN, LEFT,LEFT,LEFT,PICKUP,LEFT,LEFT,DOWN,RIGHT,RIGHT,RIGHT,RIGHT,RIGHT,DOWN, LEFT,LEFT,LEFT,LEFT,LEFT,UP,RIGHT,RIGHT,RIGHT,RIGHT,RIGHT,DOWN,LEFT,LEFT, LEFT,LEFT,LEFT,UP,UP,RIGHT,RIGHT,RIGHT,RIGHT,RIGHT,UP,UP,LEFT,LEFT,DROP; 51;2;72`
   - Maximum CPU Utilization: `28.8%`
   - Maximum Memory Utilization: `167 MB`

[1]Recall that the integers in the result string are the number of dead people, the number of retrieved boxes and the numebr of expanded nodes

3. **ID**:
   - Result: `DOWN,DOWN,DOWN,LEFT,LEFT,PICKUP,UP,UP,RIGHT,DROP,DOWN,LEFT,LEFT, LEFT,PICKUP,RETRIEVE,UP,UP,RIGHT,PICKUP,RETRIEVE,DOWN,RIGHT,RIGHT,DROP,UP; 39;2;369879`
   - Maximum CPU Utilization: `48.7%`
   - Maximum Memory Utilization: `547.6 MB`
4. **GR1**:
   - Result: `LEFT,LEFT,LEFT,DOWN,DOWN,RIGHT,DOWN,LEFT,UP,UP,UP,LEFT, DOWN,RIGHT,UP,LEFT,DOWN,RIGHT,UP,LEFT,DOWN,DOWN,RIGHT,UP,UP, LEFT,DOWN,RIGHT,UP,LEFT,DOWN,RIGHT,UP,LEFT,DOWN,DOWN,RIGHT,DOWN, LEFT,UP,UP,UP,RIGHT,DOWN,LEFT,DOWN,RIGHT,UP,UP,LEFT,DOWN,RIGHT,UP, LEFT,DOWN,DOWN,DOWN,RIGHT,RIGHT,PICKUP,RIGHT,RIGHT,DOWN,RIGHT,UP, LEFT,UP,UP,UP,LEFT,LEFT,LEFT,DOWN,LEFT,DOWN,RETRIEVE,UP,UP,RIGHT, RIGHT,DOWN,RIGHT,DROP;175;1;583`
   - Maximum CPU Utilization: `26.9%`
   - Maximum Memory Utilization: `184.4 MB`
5. **GR2**:
   - Result: `UP,LEFT,LEFT,LEFT,LEFT,DOWN,RIGHT,PICKUP,RETRIEVE,LEFT,DOWN,RIGHT, RIGHT,RIGHT,DROP,LEFT,LEFT,LEFT,DOWN,PICKUP,RETRIEVE,RIGHT,RIGHT,RIGHT,RIGHT, RIGHT,DOWN,LEFT,LEFT,LEFT,PICKUP,RETRIEVE,LEFT,LEFT,DOWN,DOWN,RIGHT,RIGHT, RIGHT,RIGHT,RIGHT,UP,UP,LEFT,UP,LEFT,UP,DROP;56;3;96`
   - Maximum CPU Utilization: `24.6%`
   - Maximum Memory Utilization: `91.6 MB`
6. **AS1**:
   - Result: `LEFT,LEFT,LEFT,PICKUP,DOWN,RIGHT,RIGHT,DROP,DOWN,DOWN,LEFT,PICKUP,UP, UP,RIGHT,DROP,DOWN,LEFT,LEFT,LEFT,PICKUP,RETRIEVE,RIGHT,RIGHT,DOWN,RETRIEVE, UP,RIGHT,UP,DROP;34;2;1395`
   - Maximum CPU Utilization: `26.8%`
   - Maximum Memory Utilization: `139.8 MB`
7. **AS2**:
   - Result: `LEFT,LEFT,LEFT,PICKUP,RIGHT,DOWN,RIGHT,DROP,DOWN,DOWN,LEFT,PICKUP, UP,UP,RIGHT,DROP,LEFT,LEFT,LEFT,DOWN,PICKUP,DOWN,RIGHT,RIGHT,RETRIEVE,UP, UP,RIGHT,DROP,LEFT,LEFT,LEFT,DOWN,RETRIEVE;34;2;3626`
   - Maximum CPU Utilization: `24.1%`
   - Maximum Memory Utilization: `127.4 MB`

The CPU utilisation of all search algorithms visually compared to each other is present in Figure 1 The CPU utilization varied between the different search algorithms but IDS scored the highest, which is expected due to the fact that it computes depth first search iteratively until it finds a solution. With regards to the memory usage, we notice that as the number of expanded nodes increases, the memory usage increases. This makes sense because, in our implementation, when we expand the nodes we store all expanded nodes in the RAM.

AS1 and AS2 both yielded the optimal number of dead people (minimum) and retrieved boxes (maximum).

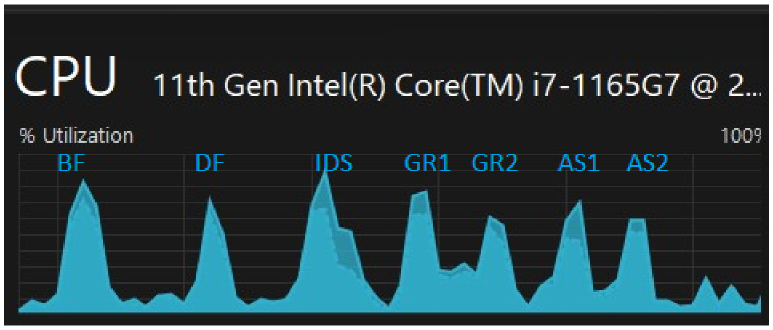

Figure 1: CPU utilizations of the search algorithms