AI Project 1

1 A brief discussion of the problem

We are a member of the coast guard force in charge of a rescue boat that goes into the sea to rescue other sinking ships. The area in the sea that we can navigate is an $(m \times n)$ grid of cells where $5 \leq m; n \leq 15$. Every action we take will have a significant impact on the other ships, therefore we'll need to develop a search algorithm that will plan the route to rescue the trapped passengers. Several ships are scattered at random locations, and each has a random initial number of passengers p, where $0 < p \leq 100$. Every action taken will, unfortunately, result in the death of a person from each trapped ship. The ship will become a wreck after all of its crew members die, but our mission does not end there; we must still retrieve the ship's black box. The black box has exactly 20 time step before the entire wreck sinks, therefore, it's critical to retrieve it to prevent adverse incidents in the future. Unfortunately, we can't always save everyone because our boat has a limited capacity c where $30 <= c < 100$, so we must drop carried-on passengers at several known rescue stations before returning for the others. Several crucial decisions must be made to save as many passengers as possible and recover as many black boxes as possible. Many search algorithms, including BFS, DFS, IDS, UC, GR, and AS, have been investigated in order to find the optimal route for the coast guard boat, which will, hopefully, result in fewer deaths and lost black boxes.

2 A discussion of our implementation of the search-tree node ADT

Our **node** class in the tree is defined with some variables which are:

- **state** which is an instance of State class which describes the current state for this node.
- **Location** is an instance of Location class to describe the location.
- **Parent Node** which is an instance of the Node class, And it indicates the parent node of the current node if there is a one.

- **Operator** is a string which describes the action that is made by the parent to give us this node.
- **extraActions** is a string which indicates the extra actions done in the node like pickup people or retrieve black box or drop people in station.
- **depth** which is an integer value indicates the depth of the node in the search tree
- **pathCoast** is an array of integer values that describes the number of deaths, number of lost boxes and a heurestic value (for greedy and AStar).

  Note : The name of this attribute is pathcost, which is not much descriptive, however it contains enough information about deaths and lost boxes. Also, it may contain the heuristic value as a third element only for greedy and AStar search strategy. Otherwise, the array contains only two values.The first value, which is for deaths, will be incremented after each action and the second one, which is for lost boxes, will be incremented after a black box becomes totally damaged.

3 The Project Structure

We implemented a class for each strategy. So, we have six different classes for the required strategies :

a) **BFS Class :** contains BFS search algorithm.

b) **DFS Class :** contains DFS search algorithm.

c) **IDS Class :** contains IDS search algorithm.

d) **UC Class :** contains UC search algorithm.

e) **Greedy Class :** contains greedy algorithm with two different heuristics.

f) **AStar Class :** contains AStar algorithm with two different heuristics.

Each one of the above classes is a child of a parent class called Search. This parent class contains all common methods between the different algorithms. Most of those methods will be discussed in the main functions description section.
Also, we have other classes that we need to implement :

a) **State Class :** This class contains the needed information about the world around the guard. These information includes an arraylist of Ships, contains all ships details such as the location and number of living passengers for each of them. Also, it includes information about Wrecks, if they exist, including their location and current damage value. In addition, we also have the current capacity that guard has at this state.

b) **Location Class :** This class consists of two integers as instance variables. Those variables are (x,y) and they will be used to help us to indicate the position of each object on the grid.

c) **Ship Class :** This class describes the ship we have in our problem. So, it contains an instance of Location class to decribe the location of each ship on the grid. it also contains number of passengers on each one and this number will be updated after each action.

d) **Wreck Class :**This class describes the wreck we have in our problem. So, it contains an instance of Location class to describe the location of each wreck on the grid. it also contains a counter value that indicates the damage value of the black box.

e) **Station Class :**This class describes the station we have in our problem. So, it only contains an instance of Location class to indicate where each station is located.

4 A discussion of our implementation of the Coast-Guard Class

This class contains two main methods. The first one is genGrid() method which is responsible for randomly generating a grid. It returns a string describing this

generated grid as described in the project description.
The second method is solve() method which accepts three parameters:

a) **String grid :** string format contains info about the grid.

b) **String strategy :** which indicates which strategy will be used to solve the problem.

c) **boolean visualize :** which is used in case if we want to visualize the execution of solution of the problem.

This method is responsible for choosing the desired strategy based on the parameter strategy. That's why, it uses switch cases for this purpose. If the strategy was "BF", then it will instantiate an object from BFS class and then calling its search method that will give us the solution and the same will be done for other strategies.

5 A description of the main functions we implemented

(1) **genGrid**: This function will initialize the grid dimensions (mXn) where 5 <=n,m <=15 in addition to, locating different cells for the coast guard boat, a random number of ships, and stations (No more than one object is occupying the same grid cell). Each ship will contain a random number of passengers to be rescued varying between 0 <p <=100. The coast guard boat can rescue a random number of passengers varying between 30 <=c <100.

(2) **GoalTest**: Will check if the current expanded node has reached the goal state by checking if all ships are destroyed (every ship turns into wreck once there is no left passenger on the ship), no available wrecks are left to retrieve its black box, and no passengers are left on my coast guard ship.

(3) **isExpanded**: That will check if the current node has already been visited/expanded or not to avoid the repeated states problem.

(4) **isShip**: This function will determine whether the current stepped-on grid cell is a ship in order to take the appropriate action as in pickUp(9).

(5) **isWreck**: This function will determine whether the current stepped-on grid cell is a wreck in order to take the appropriate action as in retrieve(10).

(6) **isStation**: This function will determine whether the current stepped-on grid cell is a station in order to take the appropriate action as indicated in drop(11).

(7) **UpdateWrecks**: This function will damage every wreck's black box by 1 after every action is taken also, handling the removal of wrecks from the grid once the black box is expired, and in this case, it will increase number of lost boxes.

(8) **UpdateShips** : This function will kill a passenger from every available ship on the grid after every action is performed, and based on that number of

deaths will be increased. It also handles the transformation from ship to wreck once all passengers die on the ship.

(9) **pickUp**: The coast guard boat will pick up as many passengers as possible additionally, handling the transformation from ship to wreck if every passenger is picked up from the ship.

(10) **retrieve**: The coast guard boat will retrieve the black box and remove the wreck from the grid.

(11) **drop**: The coast guard boat will safely drop off all the carried-on passengers at the station and set the current capacity.

(12) **expand**: This function will expand the current node by first validating the next movement action (up, down, left, right) following, the creation of a new node with all updated attributes in the new state. Subsequently, updating all available ships and wrecks of the newly created node as mentioned in UpdateWrecks(7) and UpdateShips(8). Finally, adding the newly created expanded nodes to the expanded node stack/queue/priority queue based on the search problem algorithm.

(13) **findPath**: Will determine the path from the initial state to the goal state, as specified in GoalTest(2).

(14) **visualize**: Will print a visual demonstration while navigating various steps of the discovered final route for the coast guard boat.

6 A discussion of how you implemented the various search algorithms

(1) **BFS**: This function accepts a string representation of the encoded generated gird. It begins by decoding the string representation by initializing the coast guard location and capacity, the ArrayList of all surrounding ships, and the ArrayList of all surrounding stations. The current node will be initialized first, which will contain the current coast guard boat state, which is the initial state, and its surrounding ships, as well as the queue of children nodes that must be explored, and another ArrayList of all expanded nodes that will be used to handle the repeated state issue. The algorithm will iterate on the queue until either the queue is empty or the goal state is reached. Every iteration will check whether the coast guard boat is standing on a ship, a wreck and a station, as indicated in isShip(4), isWreck(5),and isStation(6), and will take the proper action based on that. Once the proper action is made, the current node's children will be expanded, as indicated in expand(12). After it reached the goal state will print a visual demonstration, if the boolean was true, while performing various actions to reach the goal state as mentioned in visualize(14) eventually, it returns the sequence of action that was taken to reach this goal, the number of dead passengers, retrieved black boxes and expanded nodes.

(2) **DFS**: This function accepts a string representation of the encoded generated gird. It begins by decoding the string representation by initializing the coast guard location and capacity, the ArrayList of all surrounding ships, and the ArrayList of all surrounding stations. The current node will be initialized first, which will contain the current coast guard boat state, which is the initial state, and its surrounding ships, as well as the stack of children nodes that must be explored, and another ArrayList of all expanded nodes that will be used to handle the repeated state issue. The algorithm will iterate over the stack until either the stack is empty or the goal state is reached. Every iteration will check whether the coast guard boat is standing on a ship, wreck, or station, as indicated in isShip(4), isWreck(5),and isStation(6), and

will take the proper action based on that. Once the proper action is made, the current node's children will be expanded, as indicated in expand(12). After it reached the goal state will print a visual demonstration,if required, while performing various actions to reach the goal state as mentioned in visualize(14) eventually, it returns the sequence of action that was taken to reach this goal, the number of dead passengers, retrieved black boxes and expanded nodes.

(3) **UC**: This function accepts a string representation of the encoded generated gird. It begins by decoding the string representation by initializing the coast guard location and capacity, the ArrayList of all surrounding ships, and the ArrayList of all surrounding stations. The current node will be initialized first, which will contain the current coast guard boat state and its surrounding ships, as well as the priority queue of children nodes that must be explored, and another ArrayList of all expanded nodes that will be used to avoid the repeated state issue. The algorithm will iterate on the priority queue until the priority queue is empty or the goal state is reached. This priority queue implements comparableNode class which sorts the nodes based on the number of deaths, and if two nodes have the same number of deaths, then it will sort them based on number of black boxes. Every iteration will check whether the coast guard boat is standing on a ship, wreck, or station, , as indicated in isShip(4), isWreck(5),and isStation(6), and will take the proper action based on that. Once the proper action is made, the current node's children will be expanded, as indicated in expand(12). After it reaches the goal state, we will print a visual demonstration, if required, while performing various actions to reach the goal state as mentioned in visualize(14) eventually, it returns the sequence of action that was taken to reach this goal, the number of dead passengers, retrieved black boxes and expanded nodes.

(4) **IDS**: The algorithm will behave exactly like the DFS, but its depth will be limited. The depth limit is set initially to zero and if the stack becomes empty before reaching the goal state, then the limit will be increased by one, and the DFS will run again on this limit depth as indicated previously

in DFS(2) until it reaches the goal test.

(5) **Greedy**: This function accepts a string representation of the encoded generated gird. It begins by decoding the string representation by initializing the coast guard location and capacity, the ArrayList of all surrounding ships, and the ArrayList of all surrounding stations. The current node will be initialized first, which will contain the current coast guard boat state and its surrounding ships, as well as the priority queue of children nodes that must be explored, and another ArrayList of all expanded nodes that will be used to avoid the repeated state issue. The algorithm will iterate on the priority queue until the priority queue is empty or the goal state is reached. This priority queue implements comparableNode2 class which sorts the nodes based on the heuristic value that the node has. For this purpose, we used two heuristic functions, the First Heuristic(7) and Second Heuristic(8). The first heuristic function gives higher priority to rescue passengers from the nearest ship. The second heuristic function gives higher priority to rescue passengers from the heaviest ship. Both of them are admissible. Every iteration, we will check whether the coast guard boat is standing on a ship, wreck, or station, , as indicated in isShip(4), isWreck(5),and isStation(6), and we will take the proper action based on that. Once the proper action is made, the current node's children will be expanded, as indicated in expand(12). After it reaches the goal state, we will print a visual demonstration, if required, while performing various actions to reach the goal state as mentioned in visualize(14) eventually, it returns the sequence of action that was taken to reach this goal, the number of dead passengers, retrieved black boxes and expanded nodes.

(6) **AStar**: This function accepts a string representation of the encoded generated gird. It begins by decoding the string representation by initializing the coast guard location and capacity, the ArrayList of all surrounding ships, and the ArrayList of all surrounding stations. The current node will be initialized first, which will contain the current coast guard boat state and its surrounding ships, as well as the priority queue of children nodes that must be explored, and another ArrayList of all expanded nodes that will

be used to avoid the repeated state issue. The algorithm will iterate on the priority queue until the priority queue is empty or the goal state is reached. This priority queue implements comparableNodeAStar class which sorts the nodes based on some evaluation function which is :

$$f(n) = g(n) + h(n)$$

where $g(n)$ represents the path cost which is equal to the number of deaths, and $h(n)$ represents the heuristic value.

For this purpose, we used two heuristic functions, the First Heuristic(7) and Second Heuristic(8). The first heuristic function gives higher priority to rescue passengers from the nearest ship. The second heuristic function gives higher priority to rescue passengers from the heaviest ship. Both of them are admissible. So, the priority queue sorts the nodes based on that evaluation function, and if two nodes has the same evaluation function, then they will be sorted based on the number of lost boxes. Every iteration, we will check whether the coast guard boat is standing on a ship, wreck, or station, , as indicated in isShip(4), isWreck(5),and isStation(6), and we will take the proper action based on that. Once the proper action is made, the current node's children will be expanded, as indicated in expand(12). After it reaches the goal state, we will print a visual demonstration, if required, while performing various actions to reach the goal state as mentioned in visualize(14) eventually, it returns the sequence of action that was taken to reach this goal, the number of dead passengers, retrieved black boxes and expanded nodes.

7 A discussion of heuristics Functions

(7) **First Heuristic**:

This function gives an estimation cost from the current node till the node that satisfies the goal state. This estimation is based on the number of passengers that will die if we go through this path if we have ships, or number of black boxes that will be lost if we have only wrecks on the grid.

The approach is based on that the optimal solution will be achieved if we intend at each step to go to the nearest ship. So, this function first searches for the nearest ship and then calculates the heuristic value as follows :

$$\text{h(n)} = \min(\text{no.of living passengers on nearest ship,}$$
$$\text{distance to the nearest ship)}$$

Let's imagine that we have on the grid (n) number of ships, each of them has some number of passsengers :

$$(S1, P1), (S2, P2), ..., (Sn, Pn)$$

where, ($1 <= i <= n$), Si is the ship location, Pi is the number of its passengers. And let's assume that the nearest ship is $S1$ with (k) steps from the guard location, so if the guard moves to this ship, which is the nearest one, this will cost us one passenger from each ship at each step. The actual cost will be :

$$\sum_{i=1}^{n} min(P_i, k)$$

As you can see, number of passengers that will die from each ship will be the minumim value between the distance, which represents the number of actions, and number of living passengers on that ship. So, the total cost will be the summation over all ships as indicated in the above formula. However, our heurestic function will give us only the minimum value between the distance and the number of living passengers on the nearest ship. So, it will be :

$$h(n) = min(P_1, k)$$

which will be always less than or equal the actual cost :

$$min(P_1, k) <= \sum_{i=1}^{n} min(P_i, k)$$

The heuristic will be always less than the actual if we have more than one ship, and it will be equal to the actual cost if we only have one ship on the grid. So, the heuristic function never overestimates the actual cost. So, it will be always **admissible**.

So, our function always searches for the nearest ship, and when it reaches that ship, it will pick as much as the guard can, and then starts to search again for the next nearest ship. However, we have a case where the guard can not pick more passengers, and needs to go to station. So, the function, in that case, we will search for the nearest station instead of searching for the nearest ship. So, our heuristic value will be :

$$\text{h(n)} = \min(\text{number of living passengers on the whole grid},$$
$$\text{distance to the nearest station})$$

The actual cost will be the same, except the distance k , which represents now the distance to the nearest station. Assume, number of all living passengers is P. Here we have two cases. The first case is that the number of living passengers is greater than the distance. In that case, the actual cost will be the number of passengers will die, which will be more than k passengers. Our heuristic will give us k which is always less than the actual cost in this case. The second case is that number of living passengers is less than or equal the distance. So, the heuristic will be equal to the actual cost which is the number of living passengers. So, again the function never overestimates the cost and it is **admissible**.

So, the functions will continue as explained above till there are no passengers to be rescued. In this case, we start to collect the black boxes. So, this function gives priority to passengers and will save passengers first as much as it can and after that it will look for the boxes. For collecting the black boxes, we look first to nearest wreck from the current location, and again we have two case. The first case is that the timelife of the nearest black box is less than the distance to the wreck itself, so we can not retrieve this black box. In that case, our heuristic will be one, which is less than or equal to the actual cost in that case which is number of boxes that will be lost. The second case is that the nearest black box can be retrieved if we compare the distance to its damage. In this case, the heuristic will be zero, which is also less than or equal the actual cost. In both cases, the function never overestimates the cost, so it is **admissible**.

As you notice, the heuristic after finishing all ships and wrecks will be

equal to zero.

(8) **Second Heuristic**:

This function gives an estimation cost from the current node till the node that satisfies the goal state. This estimation is based on the number of passengers that will die if we go through this path if we have ships, or number of black boxes that will be lost if we have only wrecks on the grid. The approach is based on that the optimal solution will be achieved if we intend at each step to go to the heaviest ship, which is the ship that has the largest number of passengers after reaching it. To find this ship, we need first to loop over all ships on the grid and calculates the left number of passengers for each ship if we go to it. Let's imagine that we have on the grid (n) number of ships, each of them has some number of passsengers :

$$(S1, P1), (S2, P2), ..., (Sn, Pn)$$

where, ($1 <= i <= n$), Si is the ship location, Pi is the number of its passengers. To calculate the number of remaining passengers, we use the below formula :

$$remaining\ P_i = P_i - k$$

The ship that has the largest number of remaining passengers will be choosen to be reached first. The actual cost to go to this ship will be :

$$\sum_{i=1}^{n} min(P_i, k)$$

where k is the distance to the heaviest ship.

Our heuristic will be the minimum value between the distance and number of living passengers on the heaviest ship.

The heuristic will be always less than the actual if we have more than one ship, and it will be equal to the actual cost if this heaviest ship is the only one on the grid. So, the heuristic function never overestimates the actual cost. So, it will be always **admissible**.

So, our function always searches for the heaviest ship, and when it reaches that ship, it will pick as much as the guard can, and then starts to search

again for the next heaviest ship. However, we have a case where the guard can not pick more passengers, and needs to go to station. So, the function, in that case, we will search for the nearest station instead of searching for the heaviest ship. So, our heuristic value will be :

$$\text{h(n)} = \min(\text{number of living passengers on the whole grid,}$$
$$\text{distance to the nearest station)}$$

The actual cost will be the same, except the distance k , which represents now the distance to the nearest station. Assume, number of all living passengers is P. Here we have two cases. The first case is that the number of living passengers is greater than the distance. In that case, the actual cost will be the number of passengers will die, which will be more than k passengers. Our heuristic will give us k which is always less than the actual cost in this case. The second case is that number of living passengers is less than or equal the distance. So, the heuristic will be equal to the actual cost which is the number of living passengers. So, again the function never overestimates the cost and it is **admissible**.

So, the functions will continue as explained above till there are no passengers to be rescued. In this case, we start to collect the black boxes. So, this function gives priority to passengers and will save passengers first as much as it can and after that it will look for the boxes. For collecting the black boxes, we look first to nearest wreck from the current location, and again we have two case. The first case is that the timelife of the nearest black box is less than the distance to the wreck itself, so we can not retrieve this black box. In that case, our heuristic will be one, which is less than or equal to the actual cost in that case which is number of boxes that will be lost. The second case is that the nearest black box can be retrieved if we compare the distance to its damage. In this case, the heuristic will be zero, which is also less than or equal the actual cost. In both cases, the function never overestimates the cost, so it is **admissible**.

As you notice, the heuristic after finishing all ships and wrecks will be equal to zero.

First Note : The first heuristic considers the nearest ship and gives higher priority to go it. However, the second heuristic considers the heaviest ship and gives higher protity to go it. So, they are different in case you are confused between them.

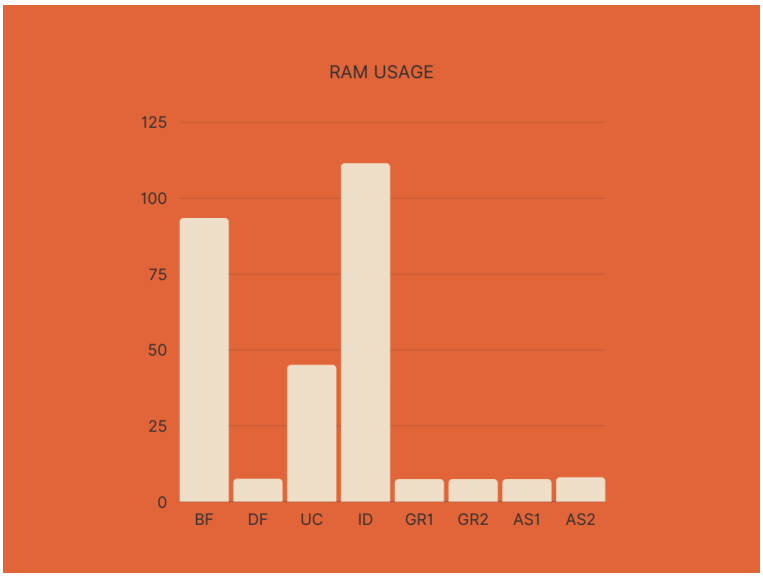
Second Note : The method we used to calculate the distance in both heuristic functions is the Manhattan distance.

8 A comparison of the performance of the different implemented algorithms

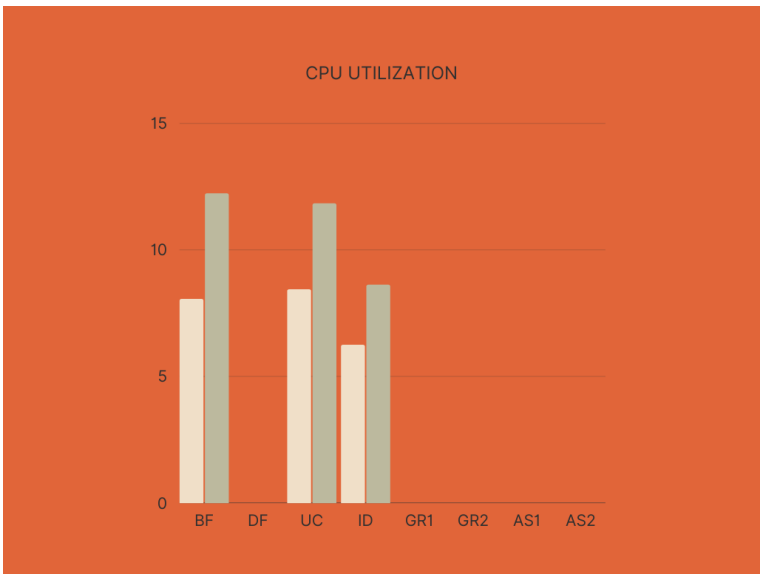
For this section, we used JProfiler to measure CPU utilization and RAM usage.

	RAM usage	CPU utilization Process Load	CPU utilization System load	Number of Expanded Nodes
BFS	93.41	8.06	12.22	91,849
DFS	7.5	0	0	801
UC	45.03	8.44	11.83	35,556
IDS	111.5	6.25	8.62	283,336
A*1	7.38	0	0	715
A*2	7.38	0	0	892
Greedy1	7.38	0	0	488
Greedy2	7.38	0	0	487

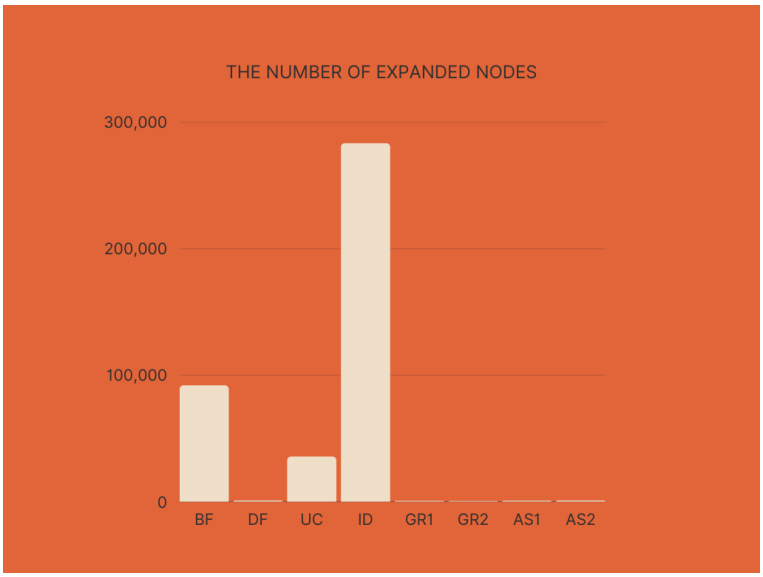
RAM Usage Graph



CPU Utilization Graph



Number of Expanded Nodes Graph



9 Some comments

As you can see from the table, the IDS has the largets number of expanded nodes and then comes the BFS. However, greedy and Astar has the lowest numbers compared to others. The same implies for cpu utilization and ram usage, IDS has the largest numbers while Astar has the lowest. Now, let's talk about the optimality and completeness for each strategy :

a) **BFS**: complete as if there is a solution , it will find it. This is because of that we have finite number of operators/actions, so number of nodes at any level will be finite, and that's why it will find the solution if there is a one. It is not optimal as it does not take into consideration path cost(number of deaths and lost boxes). However, it can be optimal under some conditions such as path cost is the depth of the node itself, then BFS is optimal as it will find the shallowest solution.

b) **DFS**: incomplete in case if we have an infinite branch in which DFS will continue going deeper and the solution may exist in another branch. not optimal as it does not take into consideration the path cost(number of deaths and lost boxes).

c) **UC**: complete as if there is a solution, then it will find it. However, if the path cost was negative, then it may result in an infinite path in which path cost decreases or never increases, and in this case it can not find the solution. Otherwise, it is complete.

d) **IDS**: complete as it visits the tree at all depths starting from depth zero, and if the solution exists, then it will find it. optimal in same cases where BFS is optimal which is that the path cost is not negative.

e) **Greedy**: complete as if there is a solution, it will find it, even if it takes a misleading path first, it will return to the right path and find the solution. It is not always optimal as the heuristic function may not have a full knowledge about the world around the guard.

f) **AStar**: complete as it will find the solution if it exists. optimal as it finds the optimal solution with the fewer number of expanded nodes.