AI Project 1 Report

Team Members

1. **Ahmed ElAmory 46-2859**
2. **Ahmed Belal 46-0642**
3. **Mohamed Sherif 46-12029**

Problem Description

You are a member of the coast guard force in charge of a rescue boat that goes into the sea to rescue other sinking ships. When rescuing a ship, you 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 you 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. You reach your goal 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. You would also like to rescue as many people as possible and retrieve as many black boxes as possible. The area in the sea that you can navigate is an m x n grid of cells where 5 <= m; n <= 15.

The grid elements are restricted to the following.

- **Coast Guard Boat**: The coast guard boat is the agent; it is the only element that can move on the grid. The coast guard can enter any cell. It has a fixed capacity 30 <= c <=> 100 which indicates the number of passengers it can carry at one time. So it may have to make multiple visits to one ship to save all the passengers on it.
- **Ships**: Each ship has a certain number of passengers and, at every time step, one of them expires. The ship is considered sunk when all of them are expired or all are picked up In that case, it becomes a wreck. Each ship contains a black box that can be retrieved after it sinks as long as it has not been completely damaged.
- **Wrecks**: Once the ship no longer has any passengers (all expire or all are picked up), it becomes a wreck. When the ship becomes a wreck (once its last passenger either expires or is picked up), in the next time step, the black box starts counting damage from 1 all the way up to 20. Once damage reaches 20, the black box is no longer retrievable.
- **Stations**: Stations are fixed and do not have any capacity limits. To count a passenger saved, they need to be dropped at a station. Initially, the grid is configured as follows.
- The coast guard boat is at a random location
- Several ships are scattered at random locations, and each has a random initial number of passengers p, where 0 < p <= 100.
- Several stations are at random locations.
- There are no wrecks.
- No two items are in the same cell.

The coast guard can perform the following actions.

- **Pick-up**: The coast guard picks up as many passengers off a ship as its remaining capacity allows. This can be done on a ship that is in the same cell as the coast guard and it only affects this ship. Once a passenger is picked up by the coast guard, they will not expire and will stay on the coast guard boat until they are dropped at a station.
- **Drop**: The coast guard drops all passengers it is currently carrying at a station. This can only be done when the coast guard and the station are in the same cell and it resets the remaining capacity of the coast guard boat to 0.
- **Retrieve**: The coast guard boat retrieves a black box. This can only be done when the coast guard boat is in the same cell as a wreck with a black box which has not been completely damaged yet. This action does not affect the coast guard's remaining capacity at all.
- Movement in any of the 4 directions (up, down, left, right) within the grid boundaries.

Problem Discussion

- The problem is a search problem, we have to find the best path to reach the goal state.
- The problem is a fully observable problem, the agent can see the whole state of the problem.
- The problem is a single agent problem, we have to find the best path for one agent to reach the goal state.
- The agent is the coast guard boat.

To takle the problem we need:-

- First we need to represent the state of the world including the agent position (i.e. The coast guard boat) at each step.
- In our problem, the state is represented by the position of the coast guard boat in the grid, the number of people on the coast guard boat and the state of the each ship in the grid (i.e. the number of alive people on the ship, is it wrecked and the damage of the blackbox).
- At every step, the agent chooses an action to do and the state changes accordingly until we reach a goal state.
- The goal state is when all the people on the ships are rescued or dead and all blackboxes are retrieved or damaged and can not be retrieved.
- We need to reach the goal state with the minimum deaths and maximum number of blackboxes retrieved.

Problem Solution

Search-Tree Node Abstract Data Type

We created a Node ADT that contains the following attributes *(Not all attributes represent the state of the world, some are used for the algorithms implemented)*:

- **parent**: The parent node of this node.
- **action**: The action that was performed to reach this node from the parent node.
- **depth**: The depth of this node in the search tree.
- **left**: The result state (node) of the acion left of this node.
- **right**: The result state (node) of the acion right of this node.

- **up**: The result state (node) of the acion up of this node.
- **down**: The result state (node) of the acion down of this node.
- **pickUp**: The result state (node) of the acion pickUp of this node.
- **dropoff**: The result state (node) of the acion drop of this node.
- **retrieve**: The result state (node) of the acion retrieve of this node.
- **x**: The x coordinate of the coast guard boat in the grid.
- **y**: The y coordinate of the coast guard boat in the grid.
- **numberOfPeopleOntheCoastGuard**: The number of people on the coast guard boat.
- **ships**: A list of ships with their states in the grid.
- **stations**: A list of stations in the grid.
- **numberOfCollectedBlackboxes**: The number of blackboxes retrieved unitl this node.
- **numberOfdeath**: The number of dead people until this node.
- **nodesExpanded**: The number of nodes expanded until this node.

The state of the problem is represented by the following attributes:

- **x**: The x coordinate of the coast guard boat in the grid.
- **y**: The y coordinate of the coast guard boat in the grid.
- **numberOfPeopleOntheCoastGuard**: The number of people on the coast guard boat.
- **ships**: A list of ships with their states in the grid.

Ship Abstract Data Type.

To represent the state of the ship we created a Ship ADT that contains the following attributes:

- **x**: The x coordinate of the ship in the grid.
- **y**: The y coordinate of the ship in the grid.
- **numberOfPeople**: The number of people on the ship.
- **wreck**: A boolean value that indicates if the ship is wrecked or not.
- **damage**: The damage of the blackbox on the ship.

Station Abstract Data Type.

To represent the station we created a Station ADT that contains the following attributes:

- **x**: The x coordinate of the station in the grid.
- **y**: The y coordinate of the station in the grid.

Search Problem Abstract Data Type

We have Generic abstract data type for search problems that contains two methods:

- **genGrid**: This method takes the number of rows and columns of the grid as input and generates a grid with the following constraints:
  - The coast guard boat is at a random location
  - Several ships are scattered at random locations, and each has a random initial number of passengers p, where 0 < p <= 100.
  - Several stations are at random locations.
  - There are no wrecks.

    - No two items are in the same cell.
- **solver**: This method takes the grid as input and solves the problem using the search algorithm that is passed to it as a parameter.

CoastGuard Problem

We have a CoastGuard class that extends the Generic class and contains the following methods:

- **genGrid**: This method takes the number of rows and columns of the grid as input and generates a string representing a grid.
- **solve**: This method takes the grid as input and solves the problem using the search algorithm that is passed to it as a parameter.
    - The method creates the initial node and calls the search algorithm that is passed to it as a parameter.
    - The method returns the solution path and the number of nodes expanded.
- **bfs**: This method takes the initial node as input and solves the problem using the breadth first search algorithm.
- **dfs**: This method takes the initial node as input and solves the problem using the depth first search algorithm.
- **ids**: This method takes the initial node as input and solves the problem using the iterative deepening search algorithm.
- **ucs**: This method takes the initial node as input and solves the problem using the uniform cost search algorithm.
- **greedy1**: This method takes the initial node as input and solves the problem using the greedy search algorithm.
- **greedy2**: This method takes the initial node as input and solves the problem using the greedy search algorithm.
- **astar1**: This method takes the initial node as input and solves the problem using the A* search algorithm.
- **astar2**: This method takes the initial node as input and solves the problem using the A* search algorithm.

Main Functions Implemented

- **getCildren**: This function is in the Node class and it takes a node as an input and returns a list of all the children of this node that can be created by performing a possible action on this node.

    - Only the childern that do not violate the constraints of the problem are returned.
    - The left node is created only if the coast guard boat is not on the left edge of the grid.
    - The right node is created only if the coast guard boat is not on the right edge of the grid.
    - The up node is created only if the coast guard boat is not on the top edge of the grid.
    - The down node is created only if the coast guard boat is not on the bottom edge of the grid.
    - The pickUp node is created only if the coast guard boat is in the same cell as a ship, the number of people on the coast guard boat is less than the capacity of the coast guard boat and there are alive people on the ship.
    - The dropOff node is created only if the coast guard boat is in the same cell as a station, the number of people on the coast guard boat is greater than 0.

  - The retrieve node is created only if the coast guard boat is in the same cell as a ship, the number of people on the ship is 0 and the ship is wrecked and the blackbox is not fully damaged.

- **handleShips** : This function is in the Node class and it is used in the getChildren function to handle updating the state of the ships after performing an action and returns a list of updated Ships to be saved in the new created child node.

  - Each action has a different effect on the state of the ships.
  - If the action is pickUp, the number of people on the ship at the same position as the coast guard boat is decreased by the number of people that the coast guard boat can pickup and one person will die from every other ship.
  - If it is any other action the number of people on every ship will decrease by 1.
  - Also, this function handles if the number of people on a ship reaches 0, the ship is wrecked and we can retrieve the blackbox.
  - The function also handles the damage of the blackbox on the ship.

- **OnAShip**: This function is in the Node class and it is used in the getChildren function to check if the coast guard boat is on a ship or not.

- **OnAStation**: This function is in the Node class and it is used in the getChildren function to check if the coast guard boat is on a station or not.

- **equals**: This function is in the Node class and it is used to check if two nodes have the same state or not.

- **hashcode**: This function is in the Node class and it is used to generate a hashcode for each node to be used in the hashset that is used in the algorithms.s

Algorithms Implementation

The general concept of the algorithms is to create a search tree and expand the nodes in the tree until the goal state is reached. The search tree is created by expanding the nodes in the tree and adding the children to the tree. The algorithms differ in the way they expand the nodes in the tree.

Breadth First Search

The Breadth First Search algorithm expands the nodes in the tree by expanding the nodes in the same level before expanding the nodes in the next level. The algorithm uses a queue to store the nodes in the tree. The algorithm starts by adding the root node to the queue. Then, it removes the first node in the queue and adds its children if this node has not been visited before (i.e. The state of the node is not repeated) to the queue. The algorithm continues until the goal state is reached or the queue is empty.

Depth First Search

The Depth First Search algorithm expands the nodes in the tree by expanding the nodes in the same level after expanding the nodes in the next level. The algorithm uses a stack to store the nodes in the tree. The algorithm starts by adding the root node to the stack. Then, it removes the first node in the stack and adds its children if this node has not been visited before (i.e. The state of the node is not repeated) to the stack. The algorithm continues until the goal state is reached or the stack is empty.

Iterative Deepening Search

The Iterative Deepening Search algorithm expands the nodes in the tree by expanding the nodes in the same level after expanding the nodes in the next level. The algorithm uses a stack to store the nodes in the tree. The algorithm starts by adding the root node to the stack. Then, it removes the first node in the stack and adds its children if this node has not been visited before (i.e. The state of the node is not repeated) to the stack. The algorithm continues until the goal state is reached or the stack is empty. The algorithm also uses a depth limit to limit the depth of the tree. The algorithm starts with a depth limit of 1 and increases the depth limit by 1 after each iteration. The algorithm stops when the goal state is reached or the depth limit is greater than the maximum depth of the tree.

Uniform Cost Search

The Uniform Cost Search algorithm expands the nodes in the tree by expanding the nodes in the same level after expanding the nodes in the next level. The algorithm uses a priority queue to store the nodes in the tree. The algorithm starts by adding the root node to the priority queue. Then, it removes the first node in the priority queue and adds its children if this node has not been visited before (i.e. The state of the node is not repeated) to the priority queue. The algorithm continues until the goal state is reached or the priority queue is empty. The algorithm uses the cost of the node as the priority of the node in the priority queue. The cost is the number of people who died from the root node to the current node and the blackboxes retrieved.

Greedy Best First Search

The Greedy Best First Search algorithm expands the nodes in the tree by expanding the nodes in the same level after expanding the nodes in the next level. The algorithm uses a priority queue to store the nodes in the tree. The algorithm starts by adding the root node to the priority queue. Then, it removes the first node in the priority queue and adds its children if this node has not been visited before (i.e. The state of the node is not repeated) to the priority queue. The algorithm continues until the goal state is reached or the priority queue is empty. The algorithm uses the heuristic value of the node as the priority of the node in the priority queue. One heuristic value is the estimation of the minimum number of people who died from the current node to the goal state. The other heuristic value is the estimation of the minimum number of people who died from the current node to the goal state and the number of blackboxes retrieved.

Astar Search

The A* Search algorithm expands the nodes in the tree by expanding the nodes in the same level after expanding the nodes in the next level. The algorithm uses a priority queue to store the nodes in the tree. The algorithm starts by adding the root node to the priority queue. Then, it removes the first node in the priority queue and adds its children if this node has not been visited before (i.e. The state of the node is not repeated) to the priority queue. The algorithm continues until the goal state is reached or the priority queue is empty. The algorithm uses the sum of the cost of the node and the heuristic value of the node as the priority of the node in the priority queue. One heuristic value is the estimation of the minimum number of people who died from the current node to the goal state. The other heuristic value is the estimation of the minimum number of people who died from the current node to the goal state and the number of blackboxes retrieved. The cost is the number of people who died from the root node to the current node and the blackboxes retrieved.

Heuristics used in Greedy Best First Search and Astar Search

Heuristic 1

The first heuristic we used was the estimation of the minimum number of people who died from the current node to the goal state.

It works by simulating the actions of the coast guard and calculating the number of deaths that will occur if we take this action.

The algorithm checks if there is capacity on the coast guard ship.

1. If there is no capacity, it searches for the nearest station. Once the station is found, the coast guard ship location is changed to be the location of the station and the number of people on the coast guard ship is set to 0, The algorithm then calculates the number of deaths that occured due to that action.
2. If there is available capacity on the coast guard ship, The algorithm searches the available ships with people on them, and then it finds the nearest ship where the coast guard can save people from it. Then it calculates the number of deaths that will occur if we take this action and changes the position of the coast guard to the position of the ship. It then calculates the number of deaths that occured due to this action and adds it to the number of deaths.

The algorithm repeats the above steps until all the people are saved or dead. It then returns the number of deaths that occured.

Heuristic 2

The second heuristic is similar to the first heuristic however it takes the number of black boxes retrieved into consideration along the number of deaths. We want to maximize the number of black boxes retrieved from the current state to the goal state and minimize the number of deaths.

It works by simulating the actions of the coast guard and calculating the number of deaths and the number of black boxes retrieved that will occur if we take this action.

The algorithm first checks if all the ships are wrecked,

1. if they are, it searches for the nearest ship with a black box that can be retrieved. Once the ship is found, the coast guard ship location is changed to be the location of the ship and the number of black boxes retrieved is increased by 1. The algorithm then calculates the damage that occured to the ships due to that action.
2. if not, the algorithm searches for the nearest ship with people on it and can be saved.
   - Once the ship is found, the coast guard ship location is changed to be the location of the ship and the number of people on the coast guard ship is increased by the number of people on the ship. The algorithm then calculates the damage that occured to the ships and the number of deaths that occured due to that action.
   - If there is no ship with people that can be saved, the algorithm looks for the nearest ship where it can retrieve a black box from after all the people in the ship died. Once the ship is found, the coast guard ship location is changed to be the location of the ship and the number of black boxes retrieved is increased by 1. The algorithm then calculates the damage that occured to the ships and the number of deaths that occured due to that action.

The algorithm repeats the above steps until all the people are saved or dead and all the black boxes are retrieved or fully damaged. It then returns the number of deaths that occured and the number of black boxes retrieved.

Results

Breadth First Search

Optimality

Breadth First Search is not otimal in our problem as it is only optimal if the cost of each action is the same. In our problem, the cost of each action is different. The cost of each action is the number of people who died from the root node to the current node and the blackboxes retrieved. Therefore, Breadth First Search is not optimal.

Completeness

Breadth First Search is complete as we always reach the goal state if it exists.

RAM Usage

Test Number	RAM Usage
1	0 MB
2	3 MB
3	6 MB
4	82 MB
5	31 MB
6	27 MB
7	6 MB
8	56 MB
9	0 MB
10	327 MB

CPU Utilization

Test Number	CPU Utilization
1	6.20%
2	7.44%
3	6.34%
4	7.21%
5	6.49%

Test Number	CPU Utilization
6	6.18%
7	6.67%
8	6.86%
9	7.64%
10	6.84%

Run Time

Test Number	Run Time
1	9.56 ms
2	30.34 ms
3	36.99 ms
4	491.85 ms
5	290.6 ms
6	196.35 ms
7	134.57 ms
8	324.5 ms
9	67.0 ms
10	1682 ms

Number of Expanded Nodes

Test Number	Number of Expanded Nodes
1	766
2	2628
3	5800
4	167977
5	97796
6	43670
7	27529
8	85961
9	12894

Test Number	Number of Expanded Nodes
10	606343

Depth First Search

Optimality

Depth First Search is not optimal because it expands the deepest nodes first which may not be the optimal path.

Completeness

Depth First Search is complete as we always reach the goal state if it exists.

RAM Usage

Test Number	RAM Usage
1	0MB
2	0MB
3	0MB
4	0MB
5	0MB
6	0MB
7	0MB
8	0MB
9	0MB
10	0MB

CPU Utilization

Test Number	CPU Utilization
1	6.85%
2	12.37%
3	11.33%
4	8.25%
5	6.85%
6	5.75%

Test Number	CPU Utilization
7	5.05%
8	6.01%
9	20.37%
10	6.87%

Run Time

Test Number	Run Time
1	4.25 ms
2	2.51 ms
3	3.24 ms
4	4.55 ms
5	4.77 ms
6	3.68 ms
7	5.77 ms
8	4.02 ms
9	4.12 ms
10	4.88 ms

Number of Expanded Nodes

Test Number	Number of Expanded Nodes
1	67
2	40
3	121
4	111
5	114
6	94
7	99
8	120
9	93
10	99

Test Number	Number of Expanded Nodes
11	115

Iterative Deepening Search

Optimality

Iterative Deepening Search is optimal only if the cost of each action is the same. In our problem, the cost of each action is different. The cost of each action is the number of people who died from the root node to the current node and the blackboxes retrieved. Therefore, Iterative Deepening Search is not optimal.

Completeness

Iterative Deepening Search is only complete if the goal state is reached in our limit of the depth.

RAM Usage

Test Number	RAM Usage
1	4 MB
2	6 MB
3	9 MB
4	198 MB
5	27 MB
6	85 MB
7	22 MB
8	84 MB
9	1 MB
10	1007 MB

CPU Utilization

Test Number	CPU Utilization
1	6.92%
2	3.72%
3	5.21%
4	6.20%
5	13.68%
6	7.43%

Test Number	CPU Utilization
7	7.25%
8	6.99%
9	13.02%
10	6.84%

RunTime

Test Number	RunTime
1	31.85 ms
2	32.99 ms
3	99.59 ms
4	1212.30 ms
5	747.02 ms
6	305.31 ms
7	187.15 ms
8	604.31 ms
9	146.79 ms
10	3757.96 ms

Number of Expanded Nodes

Test Number	Number of Expanded Nodes
1	4527
2	7073
3	31019
4	878672
5	538976
6	146391
7	82283
8	304765
9	54654
10	2597131

Uniform Cost Search

Optimality

Uniform Cost Search is optimal because it expands the nodes with the lowest cost.

Completeness

Uniform Cost Search is complete because it expands all the nodes in the search tree.

RAM Usage

Test Number	RAM Usage
1	0 MB
2	0 MB
3	1 MB
4	1 MB
5	6 MB
6	0 MB
7	1 MB
8	4 MB
9	0 MB
10	6 MB
11	32 MB

CPU Utilization

Test Number	CPU Utilization
1	5.75%
2	6.20%
3	6.19%
4	6.21%
5	14.44%
6	6.51%
7	6.84%
8	7.36%

Test Number	CPU Utilization
9	5.87%
10	7.56%
11	6.21%

RunTime

Test Number	RunTime
1	10.82 ms
2	7.24 ms
3	14.71 ms
4	87.95 ms
5	37.44 ms
6	11.81 ms
7	16.97 ms
8	36.11 ms
9	14.77 ms
10	45.71 ms
11	233.49 ms

Number of Expanded Nodes

Test Number	Number of Expanded Nodes
1	307
2	185
3	40981
4	1153
5	14691
6	4892
7	635
8	863
9	3254
10	626

Test Number	Number of Expanded Nodes
11	3872

Greedy Best First Search

Optimality

Greedy Best First Search is not optimal because it expands the nodes with the lowest heuristic value first. It does not take into consideration the cost of the path.

Completeness

Greedy Best First Search is complete because it reaches the goal state if it exists.

RAM Usage

Heuristic 1

Test Number	RAM Usage
1	0 MB
2	0 MB
3	1 MB
4	1 MB
5	1 MB
6	1 MB
7	0 MB
8	1 MB
9	1 MB
10	1 MB
11	2 MB

Heuristic 2

Test Number	RAM Usage
1	3 MB
2	6 MB
3	3 MB
4	3 MB

Test Number	RAM Usage
5	0 MB
6	10 MB
7	9 MB
8	12 MB

CPU Utilization

Heuristic 1

Test Number	CPU Utilization
1	11.58%
2	7.25%
3	7.13%
4	5.90%
5	4.65%
6	5.43%
7	8.31%
8	6.51%
9	6.22%
10	4.50%
11	6.87%

Heuristic 2

Test Number	CPU Utilization
1	6.87%
2	6.85%
3	6.29%
4	5.92%
5	6.10%
6	6.85%
7	6.87%
8	5.65%

RunTime

Heuristic 1

Test Number	RunTime
1	12.68 ms
2	6.90 ms
3	12.95 ms
4	12.40 ms
5	17.27 ms
6	26.53 ms
7	11.23 ms
8	12.66 ms
9	13.88 ms
10	19.04 ms
11	18.13 ms

Heuristic 2

Test Number	RunTime
1	27.14 ms
2	86.98 ms
3	39.32 ms
4	24.11 ms
5	80.62 ms
6	44.70 ms
7	41.08 ms
8	62.84 ms

Number of Expanded Nodes

Heuristic 1

Test Number	Number of Expanded Nodes
1	66

Test Number	Number of Expanded Nodes
2	88
3	158
4	211
5	188
6	130
7	138
8	82
9	139
10	107
11	165

Heuristic 2

Test Number	Number of Expanded Nodes
1	375
2	2529
3	415
4	340
5	1679
6	1009
7	992
8	1196

A* Search

Optimality

A* Search is optimal because it takes into consideration the cost of the path and the heuristic value. It expands the nodes with the lowest cost + heuristic value first.

Completeness

A* Search is complete because it expands all the nodes in the search tree.

RAM Usage

Heuristic 1

Test Number	RAM Usage
1	0 MB
2	0 MB
3	1 MB
4	4 MB
5	0 MB
6	0 MB
7	0 MB
8	3 MB
9	1 MB
10	3 MB

Heuristic 2

Test Number	RAM Usage
1	2 MB
2	1 MB
3	2 MB
4	6 MB
5	1 MB
6	3 MB
7	2 MB
8	5 MB
9	1 MB
10	4 MB

CPU Utilization

Heuristic 1

Test Number	CPU Utilization
1	5.92%

Test Number	CPU Utilization
2	5.91%
3	6.51%
4	7.52%
5	12.59%
6	16.30%
7	6.51%
8	20.07%
9	5.01%
10	6.85%

Heuristic 2

Test Number	CPU Utilization
1	6.52%
2	5.9%
3	7.68%
4	4.82%
5	12.4%
6	4.72%
7	6.81%
8	7.01%
9	5.92%
10	3.62%

RunTime

Heuristic 1

Test Number	RunTime
1	12.29 ms
2	8.95 ms
3	12.37 ms
4	27.36 ms

Test Number	RunTime
5	10.84 ms
6	10.10 ms
7	18.77 ms
8	25.65 ms
9	13.03 ms
10	29.75 ms

Heuristic 2

Test Number	RunTime
1	23.2 ms
2	17.86 ms
3	28.53 ms
4	60.24 ms
5	27.1 ms
6	30.04 ms
7	36.64 ms
8	44.19 ms
9	18.47 ms
10	52.96 ms

Number of Expanded Nodes

Heuristic 1

Test Number	Number of Expanded Nodes
1	61
2	101
3	199
4	375
5	77
6	88
7	65

Test Number	Number of Expanded Nodes
8	247
9	99
10	238

Heuristic 2

Test Number	Number of Expanded Nodes
1	250
2	148
3	211
4	474
5	151
6	298
7	171
8	389
9	105
10	279

Conclusion

In conclusion, we observed that as the number of expanded nodes increases, the more time it takes to run the algorithm and the more RAM it takes. Also, there are no significant difference in the CPU utilization between the algorithms.