Team 74 Report

• A discussion of your implementation of the search-tree node ADT.

Search tree node is the GeneralQueueing Abstract class in our code that every search method has to implement.

General Queueing class is the generic class and it contains: -

1. An instance variable that contains the initial node, which is the root node, we keep this variable for the IDS search strategy in which we have to keep the root node to be able to push it as we advance further in the limit deepness to push it again to the queue as it is in our expanded nodes.
2. An abstract method genQueue which generates the queue with the appropriate data structure to support our search method.
    a. In the case of BFS we use a normal ArrayDeque that uses first in last out while inserting and removing.
    b. In case of DFS and IDS we use a stack that uses FIFO so that we iterate over the tree in a deepening way, such that the first node in is the first one that gets expanded.
    c. In case of variables controlling the order of enqueuing and dequeuing which is in priorityQueueing we initialize the queue we initialize the array as the a PriorityQueue which we can control the key later on.
3. An abstract method which returns the dequeued node from the stack to be expanded. Later on the dequeues method is expanded and the children are added to the queue.
    a. In the case of BFS we dequeue by removing the first element.
    b. In DFS and IDS we dequeue by popping the element from the stack.
    c. In UC, GR and AS search we remove the highest priority element from the queue.
4. An abstract method enqueue that adds a given node to our queue.
    a. In BFS we add the node as the last element in the queue which puts it last to be expanded.
    b. In DFS and IDS we enqueue the node by adding it to the stack as it will be first element to be popped as we follow FIFO to apply deepening search.
    c. In the other search strategies which require a priority queue.
        i. In UC whenever the enqueue is called we use the compareTo method which we added to the node class which calculates the path cost to this node to determine where to put it in the queue.
        ii. In GR we use the heuristic functions depending on which heuristic we're using.
        iii. In AS we use the addition of the path cost to the heuristic function of this node to determine where to put it in the queue.

5. An Abstract method isEmpty which returns true if the queue is empty and false otherwise.

• A discussion of your implementation of the search problem ADT.

Search Problem is the generic search class to be inherited from any search algorithm.

Search Problem is an abstract class that has:

1- ArrayList of Operators: list of the available operators that can be executed.
2- Abstract goalTest(): Method that takes a state and returns a boolean of whether this state is a goal state or not
3- PathCost(): Method that takes a node and the operator to be applied on this node, the method returns the total path cost from the root to this node.
4- Abstract expand(): Method that takes a node and returns a list of all nodes that are resulted from applying the problem's operators on it.
5- Abstract getInitialState(): Abstract method that sets the initial state of the search agent and returns it.
6- GeneralSearch():
-A method that takes the name of the strategy to be used, and depending on it a class is created that corresponds to this strategy given the initial node.
-In the case of Uniform Cost, Greedy, AStar they all have one class which is "PriorityQueueing", The difference between them is how we set the node type at the initialization. To explain to the priority queue how nodes are prioritized depending on the entered strategy.
- If the Queue becomes empty we return "null" for no solution found.
- We dequeue and check if the dequeued node contains a goal state.
- If not we Expand(node) and enqueue the expanded nodes.
- In the case of Iterative deepening we check if its depth is less than or equal to the max depth before enqueueing.

• A discussion of your implementation of the MissionImpossible problem.

MissionImpossible is a child class that extends the abstract search problem class.

Mission Impossible class variables :

1. Map which is an instance for island map that generates the grid and saves the positions.
2. An array list of operators
3. Cumulative expand which save the total number of nodes expanded
4. Uniquestates which is a hashtable that saves the states that have been visited before to be able to eliminate the repeated states.

Mission Impossible class functions:

1. This class contains two constructors which are :

    a. MissionImpossible(): which generates random map , call the add operator function that will be explained and initialize the uniquestates hashtable.

    b. MissionImpossible(String grid): generates the map that is defined in the grid , calls the add operator function that will be explained and initializes the uniquestates hashtable.

2. addOperators(): function creates an instance from each operator and assigns a cost 1 for them and this cost is updated in the operators class. Those operators are then added to the array list of operators of the class.

3. goalTest(State state): this function checks if the state that is passed by the parameters passed the goal where there are no remaining IMFs in the map and Ethan position is at the submarine.

4. expand(Node node) : this function applies all the operators on a given node and then after applying each those operators we check if the returned node is already visited before or not. If yes we didn't add it to the array list of the expanded nodes but if wasn't found we added. This technique is used to eliminate the repeated states.

5. getInitialState(): this function defines an initial state to start the search tree from this state. An array of all the IMFs health and position is setted using the IMF positions and initial health.

6. getRepeated() : return the hashtable that contains the unique nodes.

7. solve(String Grid, String strategy, boolean visualize) : this function creates an instance from the mission impossible class, then starts the search by passing the search strategy to the search function finally it prints the search problem solution. Solve function is explained in details in the main functions sections

8. printsolution(Node Goal, MissionImpossible ri, boolean visualize): this function prints all the information about the path from the root node to the goal node. This function is also explained in details at the main functions section.

• A description of the main functions you implemented.

1- IslandMap class (genGrid()):

IslandMap is a class that contains information about the Island initially.
Its attributes are:

1- a 2D grid of characters to represent the map
2- Grid width
3- Grid height
4- Initial number of IMF members
5- Ethan position (stored as created cell data type)
6- Submarine position (stored as created cell data type)
7- ArrayList of IMF healths
8- Max truck count “c”

Its methods are:
1- genGrid(String grid_info): Method that takes a string representing the grid to perform the search on, the string is then split to fill the values of the attributes.

2- genGrid(): A Method that generates a random grid, the dimensions of the grid, the starting position of Ethan and the submarine, as well as the number and locations of the IMF members are randomly generated. The dimensions of the generated grid is between 5-5 and 15-15 and the number of generated IMF members is between 5 and 10. For every IMF member, a random starting health between 1 and 99 is also generated. The number of members the truck can carry “c” is randomly generated as well.

3- printGrid(): Method used to print the grid when visualize flag is set to true,
The symbols in the grid are represented as follows:

Ethan: ‘E’
IMF member: ‘I’
Submarine: ‘S’

2- solve(String Grid, String strategy, boolean visualize):

- The Method takes a string that contains information about the initial state of the map which is (grid width, grid height, Ethan (x,y) positions, submarine(x,y) position, IMF members (x,y) positions, healths, and max truck capacity), that represents the search strategy to be used.

- The Method calls MissionImpossible class with the input string, which calls IslandMap class to create the initial map.

- The method then calls the GeneralSearch() method and gives it the strategy name as input, which returns a goal node.

- Then printSolution() is called given the goal node to generate the output string which is returned, the output string contains:

  – plan is a string representing the operators Ethan needs to follow separated by commas. The possible operator names are: up, down, left, right, carry, and drop.
  – deaths is a number representing the number of deaths in the found goal state.
  – healths is a string of the format h1,...,hk where hi is the health of IMF member i in the found goal state.
  – nodes, is the number of nodes chosen for expansion during the search.

3- printSolution(Node Goal, MissionImpossible ri, boolean visualize):

- The method gets all the nodes in the route from the root to the goal node and save them into an ArrayList
- We create the output string as follows:
- We loop on every node from the root and get the applied operator, we concatenate the total number of deaths found in the goal state, We check on the health of the IMF member when he got carried by Ethan and concatenate it and finally we concatenate the number of expanded nodes.
- To print the grid if (visualize) is set to true, for every node in the goal route we extract the state information and print a grid corresponding to it following the scheme explained In IslandMap class.

• A discussion of how you implemented the various search algorithms.

- Each search algorithm inherits the abstract GeneralQueueing class that contains a constructor that initializes the initial node and contains abstract methods to be implemented in all search strategies.
- Uniform Cost, Greedy, AStar inherit the PriorityQueueing class which has a priority queue, nodes in the priority queue are prioritized using the Node class's compareTo().
- Keeping track of repeated states:
    - We create a HashSet of IslandState, IslandState has a hashCode() function that generates the hash code used. It generates a concatenated string of Ethan position, IMF members positions, and healths and converts it into an integer hashcode.
    - IslandState also has an equals method that checks if two states are equal by comparing Ethan, IMF members, Healths of both states.

- Breadth-first search (BF): we created a class that extends GeneralQueuing abstract class, We have a Queue(FIFO), we enqueue the nodes normally at the end as they are expanded to achieve level by level expansion. The class has generateQueue() to initialize the queue, enqueue() to add a node to the queue(if not null), dequeue() to remove a node from the queue/ return null if empty and isEmpty() to check if the queue is empty.

- Depth-first search (DF): we created a class that extends GeneralQueuing abstract class, We have a Stack(LIFO), to explore a single route to a leaf searching for a goal, if no goal is found it goes back to explore the next route to a leaf. Like BFS, the class has generateQueue() to initialize the queue, enqueue() to add a node to the queue(if not null), dequeue() to remove a node from the queue/ return null if empty and isEmpty() to check if the queue is empty.

- Iterative deepening Search (ID): A class that extends GeneralQueuing, It has a stack same as (DF), the difference is the on dequeuing we increase the maximum depth, clear the repeated state HashSet and enqueue the root node. And on enqueueing we check in "Search problem" class that the node's depth is not bigger than the current maximum depth.

- Uniform Cost (UC), Greedy (GRI), AStar (ASI): They are all implemented by calling the PriorityQueueing class which contains a priority queue that is used to prioritize the enqueued nodes depending on the Node type assigned when initializing the search strategy (initial_node.setType("UC") or initial_node.setType("GRi") or initial_node.setType("ASi"). i is the number of heuristics used in Greedy/AStar. We have three implemented admissible heuristics explained below.
    1. For (UC): The priority of the nodes are assigned according to the least total path cost of the node, the cost is defined in our solution as:
        - Delta death + delta damage + 15 for Carry.
        - Delta death + delta damage + 15 for Drop.
        - Delta death + delta damage + 16 for movement.
        - Where delta death is the difference between the number of deaths between the node and its parent node multiplied by 100 and by the total number of the initial IMF members count (To emphasize the importance of deaths in our problem).
        - Delta damage is the difference between the IMF damage in the current node subtracted by the IMF damage in the parent node
        - +15 is added to ensure that our first heuristic the calculated the euclidean distance between the node and the submarine position is always admissible
        - +1 for the movement is to encourage Ethan to go back to the submarine faster once there are no IMF members left on the grid or when they are already all dead.
    2. For (GR): The priority of the nodes is assigned according to the least heuristic function value which is explained in the heuristic section.
    3. For (AS): The priority of the nodes is assigned according to the least path cost and the heuristic function value summed.

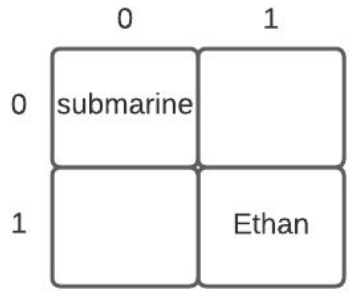
• A discussion of the heuristic functions you employed and, in the case of A*, an argument for their admissibility.

In our implementation we defined three heuristic functions. We are going to define each one and discuss its admissibility.

1. **Euclidean distance** (Heuristic 1):

First we calculated the euclidean distance between ethan position in node N1 and the submarine position as N1_euclidean. Then we calculated the euclidean distance between ethan position in node N2 and the submarine position as N2_euclidean. Then we subtracted N2_euclidean from N1_euclidean to be Delta_difference.

In the case of A* this heuristic will be admissible as we can see from the upcoming example:



As we can see here Ethan is at position (1,1) and the submarine is at position (0,0) for Ethan to reach the submarine with the defined operators he should move two steps to reach the goal but using the euclidean distance we assumed that Ethan can make a diagonal movement that is not the case so he can reach the submarine at (0,0) from (1,1) with only one movement.

Also in this heuristic function we didn't put in our account the number of deaths or the damage applied on the IMF members while the actual cost includes both the number of deaths and the damage. Even if at a certain node where Ethan has rescued all the IMF members and the damage and the deaths become zero the actual cost will be 16 for each movement operator so the actual cost will always overcome the cost of the heuristic function assuming that the maximum grid size is 15x15.

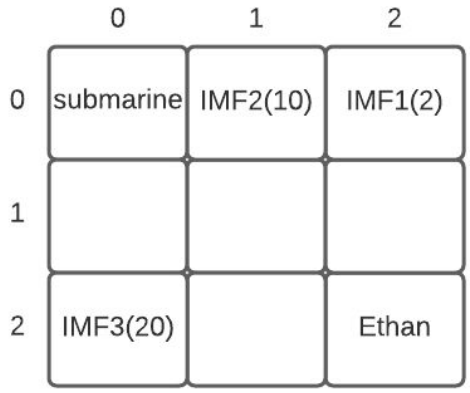
2. **Number of deaths** (Heuristic 2):

We used the number of deaths as an heuristic function. Where we calculated the number of IMF members who died till we reach node N1 as dead_N1. Then we calculated the number of IMF members who died till we reached node N2 as dead_N2. The delta_deaths is then calculated as (dead_N1 - dead_N2).

In the case of A* This heuristic will be admissible as the actual cost is a function of the number of deaths and the damage. So for any node N1 the actual cost we be the delta_damage + delta_deaths while the heuristic value will be delta_deaths only so the actual cost will always exceed the heuristic cost with no exceptions.

3. **Help who will be alive** (Heuristic 3):
   This heuristic function in a nutshell is trying to direct the search algorithm to rescue the IMf member with the least health and will still be alive when Ethan reaches him. So let's go through the implementation. For node N1 we search for the IMF member that has least health and then we calculate the number of steps it's needed to reach this IMF member if the damage applied due to those actions will kill this IMF member we will ignore him. So we finally will have the IMF member with the lowest health and that will be rescued for sure and for each ignored IMF the N1_dmage_score is incremented by 100 . And the same procedure is applied for node N2 and the N2_dmage_score is incremented by 100 when any member is ignored.

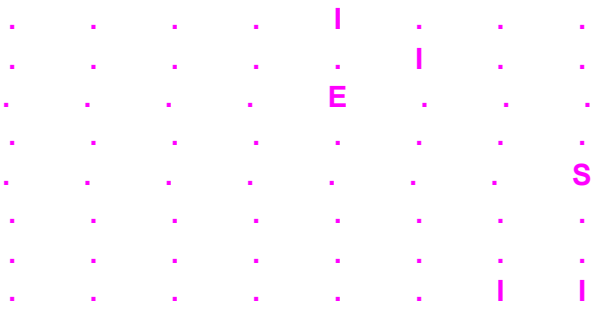
   As shown in this figure Ethan is at position (2,2) and there are 3 IMF members so Ethan will first rescue IMF2 rather than rescuing IMF1 as IMF2 can rescued as the total damage that will be applied to reach him is 8 health points so IMF2 will still have 2 health points while IMF1 will die with the first operation. Finally if all the IMF members were picked we used the euclidean distance to reach the submarine.



   In the case of A* This heuristic will be admissible as it will never overestimate the cost. As the first part of heuristic which tries to rescue the IMF members that will be alive is depending on the number of deaths the score is updated mainly based on who many IMF was ignored and died and as we mentioned above the actual cost is depending on both the deaths and the damage. While the second part of the heuristic function which is the euclidean distance part is also admissible as we mentioned in the **Euclidean distance** (Heuristic 1) section.

• At least two running examples from your implementation.

First Grid: - 8,8;4,2;7,4;5,1,7,7,4,0,6,7;93,85,72,78;1



BFS solution: -
Actions: -
Left,left,carry,down,down,down,right,right,right,right,drop,up,up,left,left,left,carry,down,down,right,right,right,drop,up,right,right,right,carry,down,left,left,left,drop,right,right,right,carry,left,left,left,drop.
Expanded nodes: - 2647
Number of deaths: - 3

DFS solution: -
Actions: -
right,right,right,right,right,left,right,left,right,left,right,left,right,left,left,left,left,left,left,left,carry,right,right,right,right,right,right,right,down,down,left,left,left,down,drop,right,right,right,carry,left,left,left,drop,right,right,right,up,carry,left,left,left,down,drop,right,right,right,up,up,left,left,left,left,left,left,carry,right,right,right,right,right,right,down,down,left,left,left,drop.
Expanded nodes: - 259
Number of deaths: - 4

IDS solution: -
Actions: -
right,right,right,right,right,down,down,carry,left,left,left,down,drop,right,right,right,carry,left,left,left,drop,right,right,right,up,up,left,left,left,left,left,left,carry,right,right,right,right,right,right,down,down,left,left,left,drop,right,right,right,up,up,left,left,left,left,left,left,left,up,carry,right,right,right,right,down,down,down,drop
Expanded nodes: - 57028
Number of deaths : - 3

UC solution: -

Actions: -
Down,left,carry,down,down,right,right,right,drop,right,right,right,carry,left,left,left,drop,up,left,up,left,left,left,up,carry,right,right,down,right,down,down,right,drop,up,right,right,right,carry,down,left,left,left,drop.
Expanded nodes: - 2411
Number of deaths: - 3

First greedy solution: -
Actions: -
Down,down,right,down,right,left,up,left,up,up,left,left,carry,right,down,right,down,right,down,right,drop,up,right,right,right,carry,left,left,down,left,drop,right,right,right,carry,left,left,left,drop,up,left,up,left,left,carry,right,down,right,down,right,drop.
Expanded nodes: - 1453
Number of deaths: - 3

Seconds Greedy solution: -
Actions: -
down,left,carry,right,right,right,down,down,drop,right,right,right,carry,left,left,left,drop,right,right,right,up,carry,left,left,left,down,drop,right,right,up,up,left,left,left,left,left,left,up,carry,down,down,right,right,right,right,down,drop.
Expanded nodes: - 934
Number of deaths: - 3

Third Greedy solution: -
Actions: -
Down,left,carry,right,right,down,down,up,left,down,right,right,drop,left,left,left,left,up,up,up,carry,right,right,down,right,down,down,right,drop,right,right,right,carry,up,up,left,left,down,down,left,drop,right,right,right,up,carry,left,left,down,left,drop.
Expanded nodes: - 880
Number of deaths: - 3

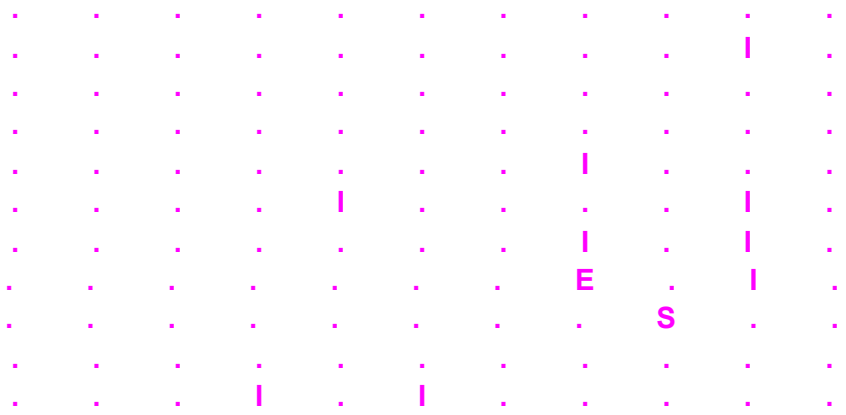
First A* solution: -
Actions: -
down,left,carry,right,down,right,down,right,drop,right,right,right,carry,left,left,left,drop,up,left,left,up,up,left,left,carry,right,right,down,right,down,down,right,drop,right,up,right,right,carry,left,left,down,left,drop
Expanded nodes: - 2409
Number of deaths: - 3

Second A* solution: -
Actions: -
down,left,carry,down,down,right,right,right,drop,right,right,right,carry,left,left,left,drop,up,left,up,left,left,left,up,carry,right,right,down,right,down,down,right,drop,up,right,right,right,carry,down,left,left,left,drop

Expanded nodes: - 2411
Number of deaths: - 3

Third A* solution: -
Actions: -
down,left,carry,right,down,right,down,right,drop,right,right,right,carry,left,left,left,drop,right,right,right,up,carry,left,down,left,left,drop,up,up,left,up,left,left,left,carry,down,down,right,right,right,right,down,drop
Expanded nodes: - 2214
Number of deaths: - 3

Second Grid: - **11,11;7,7;8,8;9,7,7,4,7,6,9,6,9,5,9,1,4,5,3,10,5,10;14,3,96,89,61,22,17,70,83;5**



BFS solution: -
Actions: -
left,left,left,carry,down,down,left,left,left,carry,right,right,right,right,carry,right,carry,right,carry,up,right,drop,up,up,up,right,right,carry,up,up,carry,down,left,left,left,left,left,carry,down,down,down,right,carry,down,right,right,drop
Expanded nodes: - 308905
Number of deaths: - 4

DFS solution: -
Actions: -
right,right,right,down,down,left,left,left,carry,right,right,right,up,up,left,left,left,left,carry,right,right,right,right,down,down,left,left,left,left,carry,right,right,right,up,up,left,left,left,left,left,left,carry,right,right,right,right,right,right,down,down,left,left,left,left,left,carry,right,right,right,up,drop,right,right,down,down,left,left,left,left,left,left,left,left,left,left,up,up,right,right,right,right,right,right,up,right,

up,right,right,right,up,carry,left,left,left,left,left,left,left,left,left,left,down,down,right,right,right,right,
right,right,right,right,right,right,down,down,left,left,left,left,left,left,left,left,left,carry,right,right,right,
right,right,right,right,right,right,up,up,left,left,left,left,left,left,left,left,left,left,up,up,right,right,right,ri
ght,right,right,right,right,right,right,up,up,carry,left,left,left,left,left,left,left,left,left,left,down,down,ri
ght,right,right,right,right,right,right,right,right,right,down,down,left,left,down,drop,right,right,down,
down,left,left,left,left,left,left,left,left,left,left,up,up,right,right,right,right,right,right,up,right,up,right,r
ight,right,up,up,left,left,left,left,left,carry,right,right,right,right,right,down,down,left,left,left,left,left,l
eft,left,left,left,left,down,down,right,right,right,right,right,right,right,right,drop
Expanded nodes: - 705
Number of deaths: - 7

IDS solution: -
Actions: -
Right,right,right,down,down,left,left,left,carry,right,right,right,up,up,left,left,left,left,carry,right,right,
right,right,down,down,left,left,left,left,carry,right,right,right,right,up,up,left,left,left,left,left,left,carry,
right,right,right,right,down,drop,right,right,down,down,left,left,left,left,left,left,left,left,left,up,carry,ri
ght,right,right,right,carry,right,right,right,right,right,up,up,left,left,left,left,left,left,left,left,left,left,up,
up,right,right,right,right,right,up,carry,right,right,right,right,right,up,carry,down,down,carry,left,left,
down,down,down,drop
Expanded nodes: - 8173781
Number of deaths : - 7

UC solution: -
Actions: -
own,down,carry,left,carry,left,carry,up,left,up,carry,up,up,right,up,carry,down,down,down,down,ri
ght,right,right,drop,left,left,left,left,left,left,down,left,carry,up,right,right,right,right,right,right,right,d
rop,down,down,right,right,up,up,up,up,up,carry,left,left,up,up,right,right,carry,down,down,left,left,
down,left,left,down,carry,down,right,right,drop
Expanded nodes: - 185589
Number of deaths: - 3

First greedy solution: -
Actions: -
down,right,down,left,carry,up,right,drop,up,right,up,right,up,up,up,carry,down,down,carry,down,d
own,left,down,left,drop,up,left,left,up,left,up,up,carry,down,down,right,down,carry,down,down,lef
t,left,left,left,left,carry,right,right,right,right,right,carry,right,up,right,drop,up,left,left,left,left,carry,ri
ght,right,right,down,right,drop,down,left,left,left,carry,right,right,up,right,drop
Expanded nodes: - 1487666
Number of deaths: - 7

Seconds Greedy solution: -
Actions: -
down,left,down,carry,left,left,left,left,left,carry,right,right,right,right,carry,left,left,up,left,up,right,rig

ht,carry,right,right,down,down,right,carry,up,right,drop,up,left,left,carry,left,up,up,up,carry,right,right,right,right,right,down,carry,left,left,left,left,left,left,left,left,left,left,down,down,right,right,right,right,right,right,right,right,down,drop,left,left,left,left,down,left,left,left,up,up,right,right,up,right,right,right,right,right,right,right,up,up,up,carry,left,left,left,left,left,left,left,left,left,down,down,right,right,right,right,right,right,right,down,down,left,down,left,drop.
Expanded nodes: - 35922
Number of deaths: - 3

Third Greedy solution: -
Actions: -
Down,left,carry,right,right,down,down,up,left,down,right,right,drop,left,left,left,left,up,up,up,carry,right,right,down,right,down,down,right,drop,right,right,right,carry,up,up,left,left,down,down,left,drop,right,right,right,up,carry,left,left,down,left,drop.
Expanded nodes: - 8191
Number of deaths: - 5

First A* solution: -
Actions: -
down,down,carry,left,carry,left,carry,up,up,left,carry,right,right,carry,right,down,right,drop,down,left,left,left,left,left,left,left,carry,right,right,right,right,up,up,up,up,up,carry,right,right,right,right,right,up,carry,down,down,carry,down,down,left,down,left,drop
Expanded nodes: - 182535
Number of deaths: - 3

Second A* solution: -
Actions: -
down,down,carry,left,carry,left,carry,up,up,left,carry,right,right,carry,right,down,right,drop,left,left,left,left,left,left,down,left,carry,right,right,right,up,up,right,up,up,up,carry,right,right,right,right,right,up,carry,down,down,carry,down,down,left,left,down,drop
Expanded nodes: - 185589
Number of deaths: - 3

Third A* solution: -
Actions: -
down,down,carry,left,carry,left,carry,left,up,up,carry,right,right,carry,right,down,right,drop,left,left,left,left,left,left,down,left,carry,up,up,right,up,up,up,right,right,right,carry,right,right,right,right,right,up,carry,down,down,carry,down,down,down,down,left,left,up,drop
Expanded nodes: - 144801
Number of deaths: - 3

• A comparison of the performance of the implemented search strategies on your running examples in terms of completeness, optimality, RAM usage, CPU utilization, and the number of expanded nodes. You should comment on the differences in the RAM usage, CPU utilization, and the number of expanded nodes between the implemented search strategies.

For the BFS search:

- **In terms of completeness the BFS is a complete search strategy as it traverses the tree level by level so it cannot miss a goal node on a better depth that another and it cannot get in an infinite loop.**
- **The optimality of BFS is not the best thing as it does not depend to optimize the number of deaths we have in our problem specially that we're allowed to pick up dead members it just finds the goal node highest in terms of depth in our search node, sometimes it might provide an optimal solution but it's just out of luck that it can change if we change the order of pushed nodes into the tree.**
- **In the bigger grid the BFS consumed over 42% of the processor memory as it expanded more than 300,000 nodes so a lot of nodes were stored at runtime consuming more than 2,100MB.**
- **In CPU utilization the BFS utilized about 2% of the processor capacity.**
- **The number of expanded nodes at the BFS was not the largest but in both grids it was definitely among the largest number of expanded nodes.**

DFS: -

- **The DFS normally is not a complete search strategy in case of an infinite tree, but we handled the repeated states to be removed so no matter what it can go very deep in the tree but it will be complete since in all examples it found a goal state.**
- **The optimality of DFS was the worst among our search strategies as usually it just goes in the first branch of the tree and it will always find a solution there that's why in both grids we find DFS with the highest number of deaths.**
- **DF almost did not consume any memory since it found a goal state with expanding the least number of nodes so it definitely consumes the least memory among our search strategies.**
- **Same goes for CPU utilization as it found the solution very quickly so the utilization is the same as the BFS.**
- **The number of expanded nodes in all provided examples were the least not ever surpassing 1,000 even in the 15x15 grid.**

IDS: -

- **IDS is a complete search strategy.**
- **The optimality of IDS was normally the same as BF as it traverses the tree level by level as well but with much more expanded nodes as it resets every time it**

reaches a depth limit, so the same as BF it just provided the lowest depth goal node in the search tree not looking for optimality.

- **The ID search so far was the highest in memory usage: and that is because of two things, the memory usage is of course high because of huge number of expanded nodes stored in the memory but technically it was not a lot more than BFS as we empty the queue everytime we reach a depth limit we just keep track of the number of expanded nodes, 2500MB memory usage, as for the CPU utilization it was very high but not as high as UC and GR and AS since everytime we enqueue and dequeue we don't just enqueue till the we find no more nodes but there are also a lot of conditions to check when performing ID search strategy.**
- **The number of expanded nodes in the IDS was definitely the largest because as we mentioned earlier it's the same as BFS it traverses the tree level by level but it always stops and resets so to reach the depth required it finishes the tree first for depth -1 and before that depth -1 and all of those nodes are re-expanded.**

UC: -

- **The UC is usually an incomplete search strategy but since we handle all repeated nodes it always gives a solution, also because we made our cost dependent on the optimality of the outcome.**
- **The UC always provided the most optimal solution for all examples.**
- **1000MB. For the UC the memory usage was not very noticeable in comparison to the BF and ID search strategies.**
- **16%. The CPU utilization on the other hand was very high at the UC as it keeps computing the path cost for every node traversed in the tree, so it performs a lot more computations while traversing the tree.**
- **The number of expanded nodes was not as high as BFS but not as low as DFS it was averages since the search keeps traversing with calculating the pathcost, it will never surpass the BFS.**

Greedy: -

- **Greedy is an informed search strategy so by logic if there is a goal greedy will get it but it does not guarantee the optimal cost., if the heuristic function is close of course.**
- **As our heuristic was admissible and it was dependent on the heuristic function so for G1 it was not optimal, G2 and G3 were very close to the optimal goal.**
- **30%. For the memory usage it depended on the number of expanded nodes for each heuristic function: the first one it always had a huge number of expanded nodes so it consumed a lot of memory, the second and the third ones on the other hand were directed to find a solution where no one dies so they expanded close number of nodes to the A* method.**
- **14%. For CPU utilization the greedy also consumes a lot since it also keeps checking and calculating heuristic function at each node before pushing it to the queue.**
- **Depending on how directed the heuristic is the number of nodes was not the highest**

A*: -

- **As for the A* it is a complete search as it will never go infinite in case of a good heuristic function if there is a goal it will find it.**
- **Since the UC provided optimal solution the A* also found the same solution but with just less number of nodes as it was more directed towards the goal.**
- **12% memory usage as the number of nodes was less than all of the other strategies except for DFS.**
- **17% Cpu utilization is more than greedy and uniform as it also has a lot of computations but it adds up the computations from greedy and the uniform cost to find an optimal goal with less number of nodes.**
- **The heuristic function was more directed towards the goal so it found the same solution as the UC search but it expanded less nodes as it was more directed.**