1. The Problem class is an abstract class that should be extended by and problem created. It contains an array of operators, initial state, and the number of expanded nodes. It also contains a number of abstract functions that any child problem should implement. These are:
   1. goalTest which returns true if the given state is a goal state.
   2. pathCost which returns an integer representing the cost of going from one state to another.
   3. stateSpace which returns a new State that represents the new state after the given operation has been applied to a given state.
   4. heuristicCost1 and heuristicCost2 where each one is a different way to calculate a heuristic cost given the current state.
   5. createSpecificNode which is used to create a node for the given problem.
   6. general\_search, depth\_limited\_search, best\_first\_search which are the general search functions that are called by the specific problem. These functions are explained in detail below in point 18.
2. MissionImpossible is a class extending the superclass Problem
3. It contains the variables gridWidth, gridHeight, submarinePos, expandedNoes, visiedStates, visitedNodes.
4. gridWidth, and gridHeight represent the width and height of the grid respectively
5. submarinePos represents the position of the submarine which is of a datatype Pos which we defined which acts as a point of the 2d map.
6. expandedNodes is an integer counter that counts the number of nodes expanded while completing the search.
7. visitedStatesHashet is a hashset used to mark which states have been visited and which states were not. It’s hashcode is a string represented as: “x y n\* s”.

* The first 2 characters “x y” represents the x and y position for Ethan which can range from 0 to 14.
* n\* is a number for each member. It can be 0,1 or 2 where 0 represents that the member is neither carried nor saved, 1 represents that the member is currently carried, 2 represents that the member has been saved. For example, if there are 3 members first 2 are saved and the last one is carried n\* becomes 2 2 1.
* s represents the sum of the members’ health whose health will not change any more such as a member who is either carried and not saved or saved and not carried.

1. Since Missionimpossible extends Problem, it must implement the following methods: goalTest, pathCost, stateSpace, heuristicCost1, heuristicCost2, createSpecificNode.
2. goalTest tests whether the goal for the problem has been reached. pathCost calculates the cost of doing a certain action. stateSpace is given a node and an operation and returns a new node that represents the current state after this operation has been applied to the given node. heuristicCost1 and heuristicCost2 calculate different heuristic costs for the given problem. createSpecificNode is used to call the constructor for the node that belongs to this problem.
3. parseGrid takes as an input a String which represents all the information needed of the game and returns an Object array where the first two elements represent the grid width and height respectively, the 3rd element represents an object of type Pos which represents Ethan’s position. The 4th element is an object of type Pos which represents the submarine’s position. The 5th element is an array of type Pos which represents each member’s position. The 6th element is an integer array that represents each members’ health. The 7th and final element is an integer which represents the truck’s capacity.
4. getPathToNode, padString, and drawGrid are helper methods for the method visualize.
5. The functions AStarSearch1 and AStarSearch2 call the generic function best\_first\_search from the Problem class with arguments AS1 and AS2 respectively.
6. UCSearch calls the generic general\_search function from the Problem class with argument ORDERED\_INSERT.
7. The functions greedySearch1 and greedySearch2 call the generic function best\_first\_search from the Problem class with arguments GR1 and GR2 respectively.
8. IDSearch calls the general function depth\_limited\_search from the class Problem given the max depth which keeps getting incremented if the solution was not found.
9. DFSearch and BFSearch both call the generic function general\_search from the class Problem with arguments ENQUEUE-AT-FRONT and ENQUEUE-AT-END respectively
10. getResultsFromNode is a simple helper function that uses the output node to create the required output string.
11. Search algorithms were implemented as follows:
    1. The solve function takes in the string grid and the solve strategy and the visualize boolean. We begin by using the grid to parse it into a MissionImpossibleNode.
    2. According to the given search strategy, a corresponding function is called that calls the correct search algorithm. There are 3 generic search algorithms implemented in the Problem class: general\_search, depth\_limited\_search, and best\_first\_search.
    3. The general search starts by creating a general queue that is given the search strategy (ENQUEUE-AT-END, ORDERED-INSERT, ENQUEUE-AT-FRONT) and creates a data structure corresponding to the correct strategy. Next, a root node is created and enqueued into the data structure. Next, we loop making sure that the queue still includes nodes doing the following:
       1. Dequeue the head of the data structure
       2. If the goal test is met then return the current node
       3. Otherwise, if the current node has the same state as an already expanded node then ignore it and go to the next node in the queue.
       4. Set the state for the current node to be visited
       5. Add the possible operations to the queue.
    4. The depth\_limited\_search does the exact same as the general search which uses the data type of “ENQUEUE-AT-FRONT” however has a limit to the depth of a node. When a node is expanded, it is first checked if its depth is larger than the maximum allowed depth, then this node is eliminated. If the depth is less than the maximum allowed depth it continues exactly like the general search.
    5. The best\_first\_search is similar to the general\_search with data type “ORDERED-INSERT” however it calculates the cost by using a heuristic function which is added to the original cost.

The first heuristic cost function relaxes the problem by removing most constraints such as carry capacity and death penalties. Each move costs 2 points (damage incurred if a member is not dead already) and we ignore that we need to spend 1 more action to carry or drop members. Basically, it estimates the cost to carry and save all members by using the shortest distance. This is done using the Manhattan distance. For example for a given state we are in, we calculate the shortest distance between Ethan and an IMF member and we save that member, then from this member, we calculate the shortest distance to the next member and so on until we have collected all members.

The second heuristic cost function is the same as the first one but with 1 more constraint which is we go to the member who is about to die first. This is done by sorting the member list according to their damage points descendingly and going to the first member in the list if the distance between Ethan and that member is small enough that if we reach him he will not die otherwise we take the second member on the list and so on. Basically giving priority to nearly dead members.

An argument for admissibility is that in our heuristic function, it only accounts for the shortest distance between the members with every cell movement costing 2 point while the actual cost is 2 \* factor (100), it is multiplied by 2 because for each movement the members take 2 points of damage also the factor 100 is added to increase the cost in the beginning of the problem to allow optimal solutions with long actions in some cases (in deeper nodes) to be expanded and checked since we stop adding costs when either no members to save or the member’s damage on the grid already reached 100. Also in our heuristics, we don’t account for the carry/drop actions. Moreover, we don’t include the death penalty. Hence our heuristic cost will never exceed or overshoot the actual cost.

The search tree node is an abstract generic class that follows the structure discussed in class.

It contains 5 attributes:

1. state: A generic object that holds the state of this node’s instance.
2. parentNode: A generic object that holds the parent node of this node’s instance.
3. operation: A string that contains the operations/actions applied to reach this node separated by a white space.
4. depth: An integer that holds the depth of this node’s instance.
5. cost: An attribute which is further split into 3 attributes namely:
   1. gCost: The actual cost from the root to this node’s instance.
   2. hCost: The estimated heuristic cost from this node’s instance to a goal node.
   3. fCost: The summation of gCost and hCost.

It contains 4 constructors:

1. A Constructor that takes 1 argument which is the Initial State of the problem.

Usually this constructor is used to create the root node in search problems which

uses search algorithms that use the actual cost or doesn’t use costs at all such as:

BFS, DFS, ID, UCS.

1. A Constructor that takes 2 arguments which are the Initial State of the problem and

hCost. Usually this constructor is used to create the root node in a search problem

which uses search algorithms that uses heuristic cost functions such as: Greedy 1,

Greedy 2, A\* 1 and A\*2.

1. A Constructor that takes 5 arguments which are the state of this node, its parent

node, the operation applied to reach this node,the node’s depth and the gCost.

1. A Constructor that takes 6 arguments, the same 5 used in the 3rd constructor +

The hCost.

**two running examples from your implementation:**

1- "6,7;4,4;6,5;4,2,5,3,6,4,5,5,6,3,4,0,1,0,0,4;15,83,38,70,43,81,3,24;2"

Depth First Search :

Completeness: Complete given our context of removing repeated states resulting in termination

Optimality: not Optimal

RAM usage:31.68 MB

CPU usage: 48%

Expanded nodes: 226

Breadth First Search:

Completeness: Complete

Optimality: non Optimal

RAM usage: 445.04 MB

CPU usage: 45%

Expanded nodes: 126404

Uniform Cost Search:

Completeness: Complete

Optimality: optimal

RAM usage: 95.34MB

CPU usage: 71%

Expanded nodes: 9771

Iterative Deepening:

Completeness: Complete

Optimality: non Optimal

RAM usage: 604.6 MB

CPU usage: 26%

Expanded nodes: 3014824

Greedy Search 1:

Completeness: Complete

Optimality: non Optimal

RAM usage: 31.56 MB

CPU usage: 56%

Expanded nodes: 239

Greedy Search 2:

Completeness: Complete

Optimality: non Optimal

RAM usage: 32.83 MB

CPU usage: 50%

Expanded nodes: 263

A Star Search 1:

Completeness: Complete

Optimality: Optimal

RAM usage: 102.74 MB

CPU usage: 73%

Expanded nodes: 9644

A Star Search 2:

Completeness: Complete

Optimality: Optimal

RAM usage: 106.25 MB

CPU usage: 70%

Expanded nodes: 9631

When reporting the number of nodes expanded by each algorithm, we found that the Iterative deepening yielded in the most nodes expanded among the other algorithms, having 3,014,824 nodes expanding due to the nature of the algorithm itself, where the Iterative deepening keeps repeating a process of depth first search iteratively yielding in expanding a node multiple times until the cutoff depth is equal to the goal node’s depth.

When reporting the RAM usage for the implemented algorithms, the Iterative Deepening and the Breadth first search resulted in maximum memory usage reaching almost 604 and 445 MB respectively which is logical given that these algorithms tend to expand nodes level by level resulting in many saved nodes while going deeper in the search tree resulting in using more memory.

While when reporting the CPU usage the UC, AStar1, and AStar reported the highest CPU usage reported at around 71.5%. Which is once again logical as they mainly take time computing the necessary nodes required for the solution leading in shorter program runtime leading in higher CPU utilization percentage.

2- “13,12;3,11;7,12;6,12,4,2,8,7,2,2,5,6,7,3,7,10;56,39,43,3,39,25,34;5”

Depth First Search :

Completeness: Complete given our context of removing repeated states resulting in termination

Optimality: non Optimal

RAM usage: 15.87 MB

CPU usage: 31%

Expanded nodes: 331

Breadth First Search:

Completeness: Complete

Optimality: non Optimal

RAM usage: 298.83 MB

CPU usage: 55%

Expanded nodes: 152451

Uniform Cost Search:

Completeness: Complete

Optimality: Optimal

RAM usage: 108.73 MB

CPU usage: 66%

Expanded nodes: 21504

Iterative Deepening:

Completeness: Complete

Optimality: non Optimal

RAM usage: 694.12 MB

CPU usage: 26%

Expanded nodes: 5880950

Greedy Search 1:

Completeness: Complete

Optimality: non Optimal

RAM usage: 45.54 MB

CPU usage: 51%

Expanded nodes: 911

Greedy Search 2:

Completeness: Complete

Optimality: non Optimal

RAM usage: 48.75.29 MB

CPU usage: 55%

Expanded nodes: 1067

A Star Search 1:

Completeness: Complete

Optimality: Optimal

RAM usage: 97.84 MB

CPU usage: 66%

Expanded nodes: 21466

A Star Search 2:

Completeness: Complete

Optimality: Optimal

RAM usage: 104.14 MB

CPU usage: 69%

Expanded nodes: 21466

When reporting the number of nodes expanded by each algorithm, we found that the Iterative deepening yielded in the most nodes expanded among the other algorithms, having 5,880,950 nodes expanding due to the nature of the algorithm itself, where the Iterative deepening keeps repeating a process of depth first search iteratively yielding in expanding a node multiple times until the cutoff depth is equal to the goal node’s depth.

When reporting the RAM usage for the implemented algorithms, the Iterative Deepening and the Breadth first search resulted in maximum memory usage reaching almost 694 and 298 MB respectively which is logical given that these algorithms tend to expand nodes level by level resulting in many saved nodes while going deeper in the search tree resulting in using more memory.

While when reporting the CPU usage the UC, AStar1, and AStar reported the highest CPU usage reported at around 66%. Which is once again logical as they mainly take time computing the necessary nodes required for the solution leading in shorter program runtime leading in higher CPU utilization percentage.