Mission Impossible AI Project Report

Team Number: 67

By: Abdelrahman Adel 40-0515
Mohamed Aboushenif 40-1465
Ali Ibrahim Elsebaie 40-2574
Mohamed Hossam 40-0260

SearchTreeNode class:

Consists of

- **state** : An array of strings consisting of : Ethan’s position, number of remaining IMF members, current number of carried IMF members, remaining health of IMF members and remaining IMF members locations.
- **parentNode** : A search tree node that consists of a reference to the parent node.
- **operator** : String that denotes the operation done to reach this node from parent node.
- **depth** : An integer that has the value of the current depth of this node.
- **costToRoot** : An integer that denotes the cost to reach the current node from the initial state.
- **AstarCost** : An integer that denotes the value of the costToRoot summed with heuristicValue.
- **heuristicValue** : An integer that denotes the value of the Heuristic value of the current node.
- **strategy** : A string that has the strategy that will be used in order to be able to have multiple conditions in compareTo to be able to compare between searchTreeNodes. For example if strategy is “UC" then the comparison between search tree nodes will be on costToRoot.
- **health** : This String is a version of the health in which the health of each member does not get removed when the member is carried unlike the one in the state.
- **carriedPositions** : A string that consists of zeros for members who are not carried and a '1' in the index of the member is carried and is used to help in stopping the decrease of health of carried members.

SearchProblem class:

It is an abstract class that can be extended by any search problem such as Mission Impossible, It contains search strategies, where each search strategy works on search tree nodes and expand these nodes, and the way of expansion differs from one strategy to another based on the way of enqueuing and dequeuing from the queue containing the nodes. Therefore SearchProblem class contains a function for each strategy and some helper functions.

MissionImpossible class:

It is a class that extends the SearchProblem class, and so by default it inherits all strategies implemented in the SearchProblem class. This class is important because it contains the stateTransition function which is responsible for node expansion as it handles all the logic for this specific problem which is MissionImpossible. It also calculates all the costs and heuristics for this specific problem.

Main functions Implemented:

genGrid: Generates a random grid with random number of IMF members and random health for each member and a random number of carries. It generates also a random location for Ethan and submarine.

stateTransition: This function implements the algorithm for expanding the node by checking available operations that can be done such as up or carry. This functions also implements the cost function and heuristic function and handles the decrease of health of each member and their deaths.

solve: Takes a grid and strategy and solves the problem of the grid based on the strategy based on the parameter, finally it takes as a parameter a boolean for visualizing the agent movement in the grid.

Search Algorithms:

DFS: We used a stack to enqueue and dequeue the nodes that are expanded, as the DFS works by using a stack. After the algorithm finds a goal, we loop on the parent states of the current states to output the string of the operators used to reach the goal state.

ID: This was the simplest algorithm, as we looped on the already implemented DFS, but with a threshold that is a counter incremented in each loop. The loop terminates once the DFS returns a solution. The number of nodes accumulated in each loop to show the real number of nodes visited, not the number returned by the last iteration.

UCS: We used a priority queue to put the nodes that are not repeated and those nodes are sorted based on costToRoot so when we remove the first node in the queue It will be the least cost node to be expanded. We also used a hash map to store the ancestors nodes to have a faster runtime when checking on them. After reaching the goal state we recursively iterate on parent nodes of the goal state and at the end, the operators, no of deaths, health of members and no of members expanded will be returned.

GR1 and GR2:

According to the greedy search strategy, it simply expands the nodes according to their heuristic value. As greedy simply reaches the goal by expanding the node with the least heuristic value iteratively until it reaches the goal state which have zero heuristic value. We have implemented two greedy functions "*Greedy1*" and "*Greedy2*", where each one of them sort the nodes according to two different heuristic functions as required to distinguish between them. Whereas, "*Greedy1*" sorts the nodes according to heuristic function 1 and "*Greedy2*" sorts the nodes according to heuristic function 2. Heuristic function 1 is simply calculated as the cheapest path from Ethan's position in the current node to all IMF members' positions plus the cheapest path from Ethan's position in the current node to the location of the submarine based on Manhattan distance. Given the equation :
h(n) = (∑ min { | x_pos_ethan - x_position_IMF(i) | + | y_pos_ethan - y_position_IMF(i) | } ***where (i) is from 1 to n***) + (min { | x_pos_ethan - x_position_submarine | + | y_pos_ethan - y_position_submarine | }) , for heuristic function 1, each node is assigned a heuristic value and according to that the "*Greedy1*" function sorts the nodes where the least heuristic value node is at the front of the queue. After that, the node with the least heuristic value will be expanded, doing the same operations on the expanded nodes until we reach the goal state. The "*Greedy2*" function does the same operations as the "*Greedy1*" function but with different heuristic values for the nodes using heuristic function 2, which is simply the cheapest path from Ethan's position in the current node to the IMF member with the least health on the grid. Heuristic function 2 equation is simply as follows : min { | x_pos_ethan - x_position_imfMemberWithTheLeastHealth | + | y_pos_ethan - y_position_ imfMemberWithTheLeastHealth | }. Both functions , "*Greedy1*" and "*Greedy2*", avoid repeated states where they don't return to any ancestor state by checking in an ancestor array list that if the node that will be expanded found then it will be removed and will not be expanded.

AS1: In this algorithm we used the same code of UCS but the sorting operation is based on the cost to root summed with the heuristic value obtained from the first heuristic function *"heuristic_function"*.

AS2: The AS2 algorithm is similar to the AS1 algorithm but the difference is that AS2 sorts the states according to the cost to root summed with the heuristic value obtained from the second heuristic function *"heuristic_function_2"*.

BFS: To apply BFS, the method takes the initial state and add it to the queue that will be traversed and then the stateTransition function is called to see what are the states that could be reached from that state. The states returned from the transition function is then tested for any repeated states, where if there are any repeated states they are not added to the queue. The state is then added to the prevStates hashtable(for better complexity while expanding nodes) as a key for checking on repeated states. when a goal state is reached or when the queue is empty the loop breaks and outputs the plan(if any) in the form of multiple operators to reach the goal state from the initial state given.

Discussion of the heuristic functions employed:

As previously mentioned, we have implemented two heuristic functions, "*heuristic_function",* which is the cheapest path from Ethan's position in the current state to all IMF members' positions plus the cheapest path from Ethan's position in the current state to the location of the submarine, and "*heuristic_function_2"* which is the cheapest path from Ethan's position in the current node to the IMF member with the least health on the grid. In the case of A*, both heuristic functions are admissible as they can never overestimate the actual cost to reach the goal. Whereas the actual cost to reach the goal is that the total damage incurred by the IMF members is minimal and the goal is that Ethan will be at the same location of the submarine with no remaining IMF members in the grid. For the first heuristic function, the output is just the Manhattan distance from Ethan's position to all IMF members positions and the submarine position which will never exceed the actual cost as it can be that Ethan has to carry the IMF member with the least health first and the highest health at the end. For example, if Ethan is at the top of the grid and the IMF member with the least health is at the bottom of the grid, the actual cost will be that Ethan must go all the way down to carry that IMF member. On the other hand, the first heuristic function doesn't keep track of the health of the IMF members so Ethan will carry IMF members on his way discarding their health and this probably ensure that the first heuristic doesn't overestimate the actual cost. According to the second heuristic function, its output is the Manhattan distance from Ethan's position to the IMF member with the least health, which also cannot overestimate the actual cost where it can be the case that when Ethan arrives to the IMF member with the least health on the grid, a lot of other IMF members will be dead except the carried IMF member. However, the actual cost must keep track of all health of the IMF members and their distances and see which IMF member must be carried first and what is the position of this IMF member near to all other IMF members or not in order to reduce the damage for other IMF members. Therefore, the second heuristic function can never overestimate the actual cost to the goal. The main reason that both heuristic functions, "*heuristic_function" and* "*heuristic_function_2",* are admissible is that both are relaxed that both doesn't have many restrictions on the operators.

Examples from our implementation:

Example1:

String grid = "5,5;1,2;4,0;0,3,2,1,3,0,3,2,3,4,4,3;20,30,90,80,70,60;3";

System.out.println(solve(grid, "AS1", false));

Output:

```
down,down,carry,left,left,carry,down,drop,right,up,up,carry,righ
t,up,up,right,carry,down,down,left,down,down,left,left,drop,drop
,right,right,right,carry,up,right,carry,left,left,left,left,down
,drop,drop,drop;2;0,6,78,74,4,0;347
```

Example2:

String grid = "8,8;4,2;7,4;5,1,7,7,4,0,6,7;93,85,72,78;1";

System.out.println(solve(grid, "DF", false));

Output:

```
right,right,right,right,right,down,down,carry,down,left,left,lef
t,drop,right,right,right,carry,left,left,left,drop,right,right,r
ight,up,left,left,left,left,down,left,left,left,up,right,right,u
p,right,right,right,right,right,up,left,left,left,left,left,left
,down,carry,right,right,right,right,right,right,down,down,left,l
eft,left,drop,right,right,right,up,left,left,left,left,down,left
,left,left,up,right,right,up,right,right,right,right,right,up,le
ft,left,left,left,left,left,down,left,up,carry,right,right,right
,right,right,right,right,down,down,down,left,left,left,drop;2;0,
51,0,62;109
```

Comparison

Example:

String grid = “15,15;5,10;14,14;0,0,0,1,0,2,0,3,0,4,0,5,0,6,0,7,0,8;81,13,40,38,52,63,66,36,13;1";

- **BFS:**
  - **Completeness:** In this example the breadth first search is complete as it reached the solution.
  - **Optimality:** This strategy is not optimal.
  - **Number of expanded nodes:** 4051
  - **RAM usage:** 2171272 Bytes
  - **CPU Utilization:** 36.8%

- **DFS:**
  - **Completeness:** In this example the Depth first search is complete as it reached the solution.
  - **Optimality:** This strategy is not optimal.
  - **Number of expanded nodes:** 2205
  - **RAM usage:** 2601672 Bytes
  - **CPU Utilization:** 33%

- **Greedy1:**
  - **Completeness:** In this example the Greedy search is complete as it reached the solution.
  - **Optimality:** This strategy is not optimal.
  - **Number of expanded nodes:** 12800
  - **RAM usage:** 2601440 Bytes
  - **CPU Utilization:** 16.9%

- **Greedy2:**
  - **Completeness:** In this example the Greedy search is complete as it reached the solution.
  - **Optimality:** This strategy is not optimal.
  - **Number of expanded nodes:** 14348
  - **RAM usage:** 2601440 Bytes
  - **CPU Utilization:** 19.7%

- **UCS:**
  - **Completeness:** In this example the UCS is complete as it reached the solution.
  - **Optimality:** This strategy is optimal.
  - **Number of expanded nodes:** 3882
  - **RAM usage:** 2171472 Bytes
  - **CPU Utilization:** 15.0%

- **AS1:**
  - **Completeness:** In this example the Astar1 is complete as it reached the solution.
  - **Optimality:** This strategy is optimal.
  - **Number of expanded nodes:** 3855
  - **RAM usage:** 2171576 Bytes
  - **CPU Utilization:** 19.4%

- **AS2:**
  - **Completeness:** In this example the Astar2 is complete as it reached the solution.
  - **Optimality:** This strategy is optimal.
  - **Number of expanded nodes:** 3882
  - **RAM usage:** 2171384 Bytes
  - **CPU Utilization:** 16.0%