Media Engineering and Technology Faculty
German University in Cairo

Mission Impossible

Introduction to Artificial Intelligence

Ahmed Fakhry 40-2273 T11
Karim Khaled 40-2242 T11
Team 143

Contents

1 Problem Description

In this problem we try to model a search agent called Ethan. The year is 1969 and the cold war is at its peak. Tensions between the United States (USA) and the Soviet Union (USSR) reached unprecedented levels. A double agent leaked the USSR's plan to the American Central Intelligence Agency (CIA). As the stakes could not be higher, the CIA turned to its most renowned agent and the leader of the Impossible Missions Force (IMF), Ethan Hunt. Ethan flied to an undiscovered island, however their helicoper malfunctioned and everyone had to jump out. Everybody except Ethan is badly injured and is about to die. It is Ethan's mission now to bring all IMF members to the submarine using a truck that has a fixed capacity. The island they crashed is a grid that consists of m x n cells where m,n are between 5 and 15. Each cell either contains Ethan (with the truck), an ally or the submarine. Ethan can **move** all in all directions across the grid, can **carry** IMF members and can only **drop** them at the submarine. Using search algorithms we are going to device a solution to help Ethan save the IMF members. This solution will be constructed using a variety of search algorithms which are:

- Breadth-first search.
- Depth-first search.
- Iterative deepening search
- Uniform-cost search
- Greedy search with two heuristics
- A Star search with two admissible heuristics

Implementation Language: Java

2 Search Tree Node

The generic Search Tree Node is implemented as the following 6-tuple:

- State: A **String** that holds the current state of the problem, which can be a grid or any other search problem state;
- Parent: A reference of type **Node** to the parent of this object.
- Operator: An Enum called Operator which has a list of valid Operators.

- Depth: The depth level of this node, or in the other words, the number of operators applied to reach this node.
- PathCost: The cost of this **Node** after applying the **Operator**
- TotalCost: The total cost of the path from the root **Node** until this current **Node**.

3 Search Problem

The generic Search Problem is implemented as the following 6-tuple:

- getIntialState: Returns the initial **Node** of the search problem.
- goalTest: Takesa node as an input and returns a **Boolean** indicating if the node contains the goal state.
- getAllowedOperators: Takes a node as an input and returns a list of the allowed operators on this node.
- getCost: Takes the start **Node**, the **Operator** applied and the destination **Node** in order to return the cost.
- getNextState: Takes a **Node** as an input and the **Operator** and returns the new Node.
- getOptimalRepresentation: Takes a **Node** and the a Search Strategy and returns the optimized String representation. This is used to save space when storing repeated states.

4 Mission Impossible Problem

The **MissionImpossible** class extends **SearchProblem** and implemented all of the methods discussed in the **Search Problem**.

4.1 Constructor

The constructor of the **MissionImpossible** subclass takes as input the initial **Node** of the problem and assigns a variable to this **Node**.

4.2 Mission Impossible State

The Mission Impossible state is a string that consists of:

- m x n
- Ethan X, Ethan Y
- Submarine X, Submarine Y
- IMF X, IMF Y
- Damage
- Truck Capacity

4.3 Initial State

The initial state of the game is the input grid in addition to zeros and ones to indicate which IMF member is currently carried by Ethan and which is dropped in the submarine. I.E: The state:
5,5;2,1;1,0;1,3,4,2,4,1,3,1;54,31,39,98;2
becomes
5,5;2,1;1,0;1,3,4,2,4,1,3,1;54,31,39,98;2;0,0,0,0;0,0,0,0
where the first 0,0,0,0 indicate that none of the IMF member are carried and the second 0,0,0,0 indicate that none of the IMF members are dropped in the submarine.

4.4 Operators

The Mission Impossible problem has the following possible operations:

- LEFT
- RIGHT
- UP
- Right
- Carry
- Drop

Each of the operations take Ethan 1 time unit to execute which leads to any IMF member who is not carried by Ethan, since Ethan knows how to stop the bleeding, or any IMF member member who is not in the submarine to take 2 points of damage. An ally dies at 100 damage and on longer takes any damage. It is still Ethan's job however to save even dead IMF members.

The **LEFT,RIGHT,UP,DOWN** operators can only be executed if:

- Ethan will not exceed the grid's dimensions.

The **Carry** operator can only be executed if:

- Ethan has space in the truck
- Ethan is standing on the same cell with an IMF member
- Ethan is not standing on the submarine.
- The IMF member is not already carried or already dropped.

The **Drop** operator can only be executed if:

- Ethan is carrying at least 1 IMF member.
- Ethan is standing on the submarine cell.

If an IMF member is being carried and Ethan moves, the IMF member moves.

4.5 Node Expansion

After calling **getAllowedOperators** on a **Node**, a list of valid Operators for the current state full-filling the above restrictions is returned. getNextState is then called on the node and one of the Operators returned. getNextState calls getCost on the Node before returning it to assign it a cost given the changes in the state and action performed.

- ei,ej: Ethan Postion.
- x,y: Position of an IMF member
- h: Damage taken by an IMF memberr
- c: Truck capacity left.
- cr: Carry array. See 4.3
- dr: Dropped array. See 4.3

Previous State	Operator	Next State	Note
(ei,ej,x,y,h,c,cr,dr)	LEFT	(ei,ej-1,x,y,h-2,c,cr,d)	No IMF is carried*
(ei,ej,x,y,h,c,cr,dr)	LEFT	(ei,ej-1,x,y-1,h-2,c,cr,d)	An IMF is carried*
(ei,ej,x,y,h,c,cr,dr)	UP	(ei-1,ej,x,y,h-2,c,cr,d)	No IMF is carried*
(ei,ej,x,y,h,c,cr,dr)	UP	(ei-1,ej,x-1,y,h-2,c,cr,d)	An IMF is carried*
(ei,ej,x,y,h,c,cr,dr)	RIGHT	(ei,ej+1,x,y,h-2,c,cr,d)	No IMF is carried*
(ei,ej,x,y,h,c,cr,dr)	RIGHT	(ei,ej+1,x,y+1,h-2,c,cr,d)	An IMF is carried*
(ei,ej,x,y,h,c,cr,dr)	DOWN	(ei+1,ej,x,y,h-2,c,cr,d)	No IMF is carried*
(ei,ej,x,y,h,c,cr,dr)	DOWN	(ei+1,ej,x+1,y,h-2,c,cr,d)	An IMF is carried*
(ei,ej,x,y,h,c,cr,dr)	CARRY	(ei,ej,x,y,h-2,c-1,cr+1,d)	Set Carry to 1**
(ei,ej,x,y,h,c,cr,dr)	DROP	(ei,ej,x,y,h-2,c,cr-1,d+1)	Set Carry to 0 Set Drop to 1***

Table 1: *: h-2 only applies to IMF members who are not dead (100), not carried and not dropped. **: The corresponding index of the carried IMF in the Carried (cr) array is set to 1.
***: The corresponding index of the carried IMF in the Carried (cr) array is set to 0 and in the Dropped (dr) array set to 1.

5 Generic Search

5.1 Breadth First

The breadth first search is implemented using a LinkedList queue. The first step is a while (!queue.isEmpty()) condition to keep the loop going until we longer have any children to expand. We then specify a Node current and remove the first node in the queue and assign it as the current Node. If goalTest returns true we then return it's path as the soltion, if not we continue the loop. getAllowedOperators is then called on this Node and an array of Operators is returned. Next, we call getNextState on the current Node and each of the Operators returned. We check if the returned Node from getNextState has already been visited or not by checking the HashSet of visited states. If it was not visited then we add it to the queue and add it to the visited HashSet.

5.2 Depth First

The Depth First search is implemented using a stack. Similar to Breadth First, we enter a loop until the stack is empty. We pop the top of the stack and assign it as the current Node. We then check if the current Node is the goal by calling goalTest on the Node. If goalTest returns true we then return it's path as the soltion, if not we continue the loop. getAllowedOperators is then called on this Node and an array of Operators is returned. Next, we call getNextState on the current Node and each of the Operators returned. We check if the returned Node from getNextState has already been visited or not by checking the HashSet of visited states. If it was not visited then we add it to the stack and add it to the visited HashSet.

5.3 Iterative Deepening

Iterative Deepening search starts with a depth of 1 and enters an infinite loop. The depth is incremented by iteration. We call a method names DLS (Depth Limited Search) which takes the initial state and the depth limit. Depth Limited Search first checks if the input node is already visited and returns null if true.We then call goalTest on the input node and return the node if goalTest is true. If the current node is not the goal we call getAllowedOperators to get a list of valid Operators and then call getNextState on each Operator. We check if the node returned by getNextState is not visited and then we recurisvely call DLS again decrementing the current depth and with the same maxDepth. We then add the node to the visited nodes HashSet. DLS returns null if the current depth = 0 and we did not find a goal and then DLS is called again with a greater depth and the loop continues until we find a solution. For test timeout reasons in testd5 and testd6, we have increased the depth increment to +5.

5.4 Uniform Cost

Uniform cost uses a Priority Queue which takes as input a UniformCostComparator to sort the queue. UniformCostComparator compares the cost of Node s1 and Node s2 by extending the Comparator class and returning s1.getTotalCost()-s2.getTotalCost(). As mentioned earlier, the getTotalCost function returns the cost from the root. By how a comparator works, if the result of this operation is negative then s1 is better since it has lower cost from the root and if the result is positive then s2 is better since it has a lower cost. The priority queue then sorts the elements automatically. We check if the current node is visited and skip the current loop iteration if so. If the node is not visited we check the goalTest and return it as the solution if goalTest is true. Similar to all other searches, we call getAllowedOperators and getNextState to get the children of the node and then enqueue them. The loop continues until the queue is empty.

5.5 Greedy

Greedy search works exactly the same as the Uniform Cost implementation but takes a different Comparator as a priority queue input, the GR1 and GR2 comparators. These comparators contain the heuristic function needed to calculate weight of each node.

5.6 A*

A* search works exactly the same as the Uniform Cost/Greedy search but takes a different Comparator as a priority queue input, the AS1 and AS2 comparators. These comparators contain the heuristic function needed to calculate weight of each node and then it returns the total cost of each node + the heuristic function. F(x) = g(x) + h(x) were G is the cost from the root (getTotalCost) and h(x) is the heuristic comparator.

6 Mission Impossible Search

This section discusses specific search techniques related to the Mission Impossible problem.

6.1 Repeated States Optimization

As stated in an earlier section, we defined Node as a 6 tuple which includes an additional getOptimalRepresentation function which is not in the lecture. This function optimizes the repeated state string deepening on the current

strategy. It removes health changes in the state string to avoid repeating moves (UP,DOWN,UP,DOWN,RIGHT,LEFT,RIGHT,LEFT). This causes more repeated states and drastically improves the search performance by not going around in a loop.

6.2 Goal Test

The goal test of the Mission Impossible problem checks the dropped array in the representation and returns true if the whole array is full of ones. Example: 5,5;2,1;1,0;1,3,4,2,4,1,3,1;54,31,39,98;2;0,0,0,0;1,1,1,1;
which implies that all IMF members were dropped. For performance efficiency reasons this is only checked if the Operator applied on the Node is the Drop Operator as Ethan can never win the game without the last operation being a Drop operation.

6.3 Cost Function

The get cost function we use is : The amount of dead IMF members multipled by 2 + The amount of allies which are not saved (dropped or carried). This results in very optimal paths as the Ai agent is punished for killing IMF members primarily and then secondly is punished by the amount of allies taking damage (not dropped or carried). This times 2 factor ensures that the deaths penalty is prioritized over the damage penalty. For performance reasons this is only called when when the Strategy is Uniform Cost or A*.

6.4 Greedy 1 Heuristic

The greedy 1 heuristic cost function is the sum of damages taken by allies which are not saved or carried. This value approaches 0 the closer Ethan is to the goal since saving all IMF members results in the damage taken by all non saved members to be 0.

6.5 Greedy 2 Heuristic

The greedy 2 heuristic cost function is the amount of dead IMF members which are not carried or dropped. This value approaches 0 the closer Ethan is to saving all IMF members since there will be no one who is dead and is not saved when Ethan saves all members (the goal).

6.6 A* 1 Heuristic

The A*1 heuristic cost function is similar to the Greedy 1 heuristic since it is already admissible (under estimates and approaches 0 at the goal). It is the amount of damage taken by members which are not saved. This value is then added to the total cost from the root g(x) since f(x) = g(x) + h(x)

6.7 A* 2 Heuristic

The A*2 heuristic cost function is the amount of carried allies which do not have the maximum damage taken in the grid. This heuristic is admissible because it always under estimates the cost to the goal and is 0 at the goal since the goal will be when Ethan picks up the most damaged non saved ally in the grid and drops him in the submarine.

6.8 Observations

Some of the observations we noticed while running these searches and heuristics. The greedy algorithm always expands less or equal nodes as the A* search. A* search expands less nodes than UC. In some very few cases A* expanded slightly more nodes (10 to 20). It might have to do with Java's tie breaking strategy as it was not handled.

7 Main Methods

7.1 Generating the Grid

The genGrid method generates a random grid with dimensions ranging from 5 to 15 in both rows and columns (m x n). The amount of IMF members is set randomly between 5 and 15. The submarine location, IMF positions and Ethan's starting position are then set randomly between the generated m and n dimensions. An overlap method is called to avoid overlapping any of the IMF members, ethan or the submarine in the came position. The capacity is set randomly between 1 and 10 as well.

7.2 Visualize

A visualize method generates a user interface using Java Swing. The method takes as input the initial state and the plan and draws every move on the UI every 1 second. The UI is

done by creating a 2d array of JTextFields and then adding the relative information (Ethan, IMF, Submarine) to the JTextField then adding all of the JTextFields in a GridLayout. The entire grid is redrawn as a loop goes through the solution plan input and visualization is done until Ethan saves all IMF members.

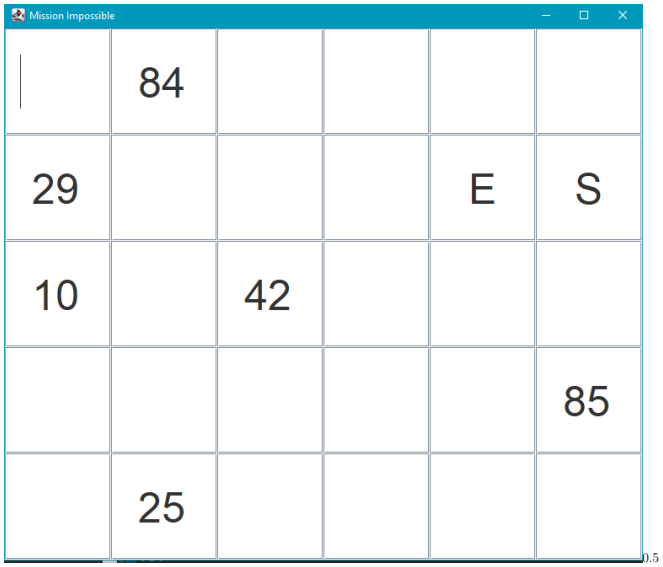

Figure 1: Visualize UI

8 Performance Comparison and Examples

8.1 Breadth First

Input Grid: 10,10;6,3;4,8;9,1,2,4,4,0,3,9,6,4,3,4,0,5,1,6,1,9;97,49,25,17,94,3,96,35,98;3

Output:right,carry,left,left,left,down,down,down,carry,left,up,up,up,up,up,carry,right,right, right,right,right,right,right,right,drop,left,left,left,left,up,carry,up,carry,up,up,right, carry,right,right,right,down,down,down,down,drop,left,left,up,up,up,carry,right,right,right, carry,down,down,carry,left,down,drop;6;100,100,55,100,96,63,100,100,100;4240285

Expanded Nodes: 4240285
Deaths: 6
Running time: 42 seconds
Breadth first is clearly non optimal as it nearly killed all IMF members.

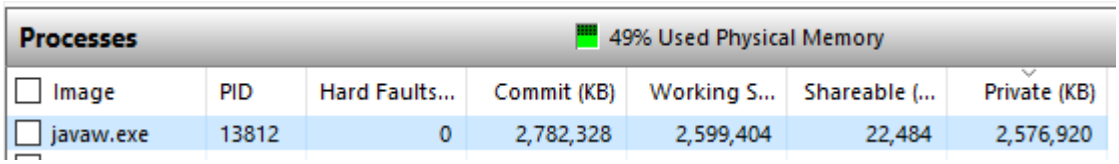

Figure 2: CPU USAGE: 15%

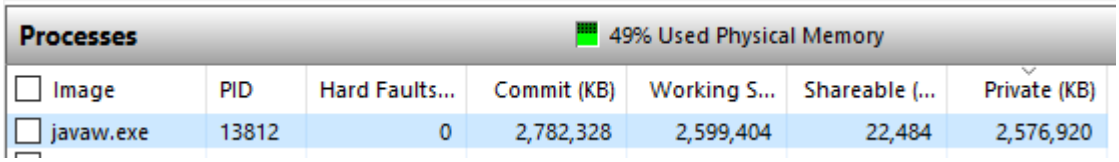

Figure 3: RAM USAGE: 2.6GB

8.2 Depth First

Input Grid: 10,10;6,3;4,8;9,1,2,4,4,0,3,9,6,4,3,4,0,5,1,6,1,9;97,49,25,17,94,3,96,35,98;3

Output:down,down,down,right,right,right,right,right,right,up,up,up,up,up,up,carry,down,down,
down,down,down,down,left,left,up,up,up,up,up,up,up,right,up,right,carry,down,down,down,down,
down,down,down,down,left,left,up,up,up,up,up,up,up,up,up,left,left,carry,down,down,down,down,
down,down,down,down,down,right,right,right,right,up,up,up,up,up,left,drop,down,down,down,down,
down,left,left,up,up,up,up,up,up,right,up,right,right,up,up,left,left,left,down,carry,down,
down,down,down,down,down,down,down,right,right,right,up,up,up,up,up,left,drop,down,down,down,
down,down,left,left,up,up,up,up,up,up,right,up,right,right,up,up,left,left,left,down,left,down,
left,carry,down,carry,down,down,down,carry,down,down,down,right,right,right,right,right,up,up,
up,up,up,up,left,drop,down,down,down,down,down,left,left,up,up,up,up,up,up,right,up,right,right,up,
up,left,left,left,down,left,down,left,down,down,down,down,down,down,down,left,left,up,up,up,up,
up,up,up,up,up,left,left,down,down,down,down,carry,down,down,down,down,down,right,carry,right,
right,right,right,right,right,right,right,up,up,up,up,up,left,drop;8;100,100,100,47,100,100,100
,100,100;1491

Expanded Nodes: 1491
Deaths: 8
Running time: 0.084 seconds
Depth first is clearly non optimal as it nearly killed all IMF members.
CPU Usage and Ram usage are impossible to measure as it finishes almost immediately. It can be observed that depth first is extremely fast when needing to find a non optimal path from a start node to a goal node. It also expands a very low amount of nodes.

8.3 Iterative Deepening

Input: 6,6;1,1;3,3;3,5,0,1,2,4,4,3,1,5;4,43,94,40,92;3
Output: up,carry,right,right,right,right,down,carry,left,down,carry,left,down,drop,right,right,carry,left,left,down,carry,up,drop;2;36,45,100,80,100;96549
Expanded Nodes; 96549
Deaths: 2
Running time: 1 second.
The ID example was ran on a smaller grid than the rest as ID takes a very long time to find a solution relative to other examples. However, we test the BF algorithm on the same 6x6 grid and came to the following observations: Due to the nature of how ID works, if there is a solution at depth X that BF search found then ID search will find the same solution or a shallower solution: BF on 6x6 Output: up,carry,right,right,right,right,down,carry,left,down,carry,left,down,drop,right,right,carry,left,left,down,carry,up,drop;2;36,45,100,80,100;19710 Expanded 19710. Running time 0.5 second. ID search also expands more nodes and takes more time to run and uses less memory than BFS.

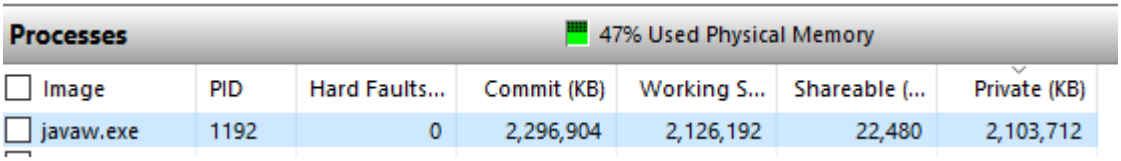

Figure 4: ID USAGE: 2.3GB

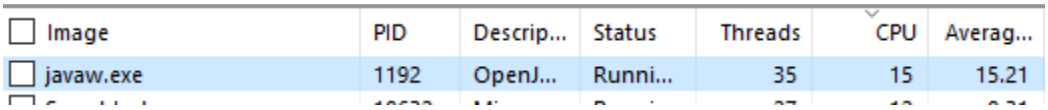

Figure 5: ID CPU: 15%

8.4 Uniform Cost

Input: 6,6;1,1;3,3;3,5,0,1,2,4,4,3,1,5;4,43,94,40,92;3
Output: up,carry,right,down,right,right,right,down,down,carry,left,left,down,carry,up,drop,right,down,right,up,up,up,carry,down,left,carry,left,down,drop;2;22,45,100,66,100;685
Expanded Nodes; 685
Deaths: 2
Running time: 0.058.

Uniform cost sucessfully provided an optimal path to the goal. It can be seen from the path that it did not waste time and automatically calculated the most optimal path to any ally. This was done by adding a penalty to each dead ally and then adding a lower penalty to damage taken which causes UC to avoid deaths then avoid damage, reaching the most optimal path. Ofcourse UC would expand less nodes than BF,DF and ID and take less time as it is an optimal search. The CPU/Ram usage could not be compared due to the lightning fast speed the test finished in. It should also be noted that due to the cost function implemented, UC would never pick a dead ally over a non dead ally as can be seen in the path output.

8.5 Greedy1

Input: 6,6;1,1;3,3;3,5,0,1,2,4,4,3,1,5;4,43,94,40,92;3
Output: up,carry,down,down,down,right,right,down,carry,left,up,right,down,left,left,down,right,
right,up,down,right,right,up,down,up,down,up,down,up,down,up,down,up,down,up,down,up,down,left,
up,up,up,carry,right,down,up,up,up,down,left,down,left,down,drop,down,down,right,right,up,up,
carry,up,up,carry,down,down,down,left,left,up,drop;3;100,45,100,56,100;1370

Expanded Nodes; 1370
Deaths: 3
Running time: 0.084s.
Greedy is a non optimal path so it resulted in a worse path than UC, which is an optimal algorithm. It also expanded more nodes than UC and took more time to find a path. It was not possible to measure CPU and RAM due to run time.

8.6 Greedy2

Input: 6,6;1,1;3,3;3,5,0,1,2,4,4,3,1,5;4,43,94,40,92;3
Output: up,carry,right,down,down,down,down,down,right,up,carry,down,right,right,up,up,carry,up
,up,left,down,down,down,left,up,drop,down,down,right,right,up,up,up,left,carry,down,down,down
,right,up,up,up,up,carry,down,down,down,left,left,up,drop;2;36,45,100,60,100;335

Expanded Nodes; 335
Deaths: 2
Running time: 0.064s.
Greedy is a non optimal path so it resulted in a worse path than UC, which is an optimal algorithm. This heuristic however did not cause the Greedy algorithm to expand more nodes than UC. It resulted in the same deaths but more damage overall. It was not possible to measure CPU and RAM due to run time.

8.7 AS1

Input: 6,6;1,1;3,3;3,5,0,1,2,4,4,3,1,5;4,43,94,40,92;3
Output: up,carry,down,down,down,right,right,down,carry,right,down,right,up,up,carry,left,left, drop,left,down,right,down,right,up,up,right,up,up,carry,down,down,left,down,left,up,drop,down, down,right,right,up,up,up,left,carry,down,down,down,left,left,left,up,up,up,right,right,down, drop;2;32,45,100,56,100;756

Expanded Nodes; 756
Deaths: 2
Running time: 0.108s.
AS1 Resulted in an optimal cost as did the UC algorithm. The paths are different but the summation of the damage and deaths is still the same, which proves that the AS1 Heuristic is admissible since it resulted in an optimal path. It should also be observed that greedy expanded less nodes than A* and that UC expanded slightly less nodes than A*. Both result in damage of 333 and deaths of 2. A* however took more running time to find a path.

8.8 AS2

Input: 6,6;1,1;3,3;3,5,0,1,2,4,4,3,1,5;4,43,94,40,92;3
Output: up,carry,right,down,right,right,right,down,down,carry,left,left,down,carry,up,drop, right,down,right,up,up,up,carry,down,left,carry,left,down,drop;2;22,45,100,66,100;685

Expanded Nodes; 685
Deaths: 2
Running time: 0.111s.
AS2 resulted in the same optimal path of UC. It expanded the same amount of nodes as well and has the same deaths of 2 and the same damage of 333.