AI Project 1 Team 45 Report

Omar Raed Elfatairy, 40-9247, T-27

Youssef Wally, 40-7641, T-36

Youssef Ramadan, 40-7266, T-29

Youssef Abobakr, 40-14040, T-23

# **SearchProblem Class:**

The search problem class is an **abstract** class that mirrors the general attributes in any given search problem. The attributes of this class are: a queue, a stack, and a priority queue to be utilized by the different search strategies, and it has a hashmap to store repeated states.

Since the order of node expansion in any given search problem depends solely on the order in which the nodes have entered the queue, any general search problem would be able to implement breadth first search, depth first search, and iterative depth first search. On the other hand, to design a path cost function and a heuristic function, one needs more information about the problem. Thus, the search problem class does not include the uniform cost method, the greedy methods, and the A star methods.

Other than the core methods, this class also has a few helped methods. It has the solve method that takes a problem in the form of a string, takes the strategy to be used to solve this problem, and a specification of whether or not a visualization of the solution is needed. It also has the isGoalState method. This method always returns false because we do not have a specification of the problem to determine what the solution is.

**You must be wondering why are the methods not abstracted?** The reasoning behind our decision to create the methods is because the children of the SearchProblem would need to have static methods in order to bypass the need to pass down a searchProblem to the nodes. This is crucial to save memory space and speed up the processes.

# **Node Class:**

In our implementation we designed the node in such a way that no reference to the children is needed. Only a pointer to the parent is needed. This is due to the fact that we are not interested in the children of a dead end, thus whenever our solution reaches a dead end in one of the branches, the Java garbage collector would handle the deletion process. Furthermore, without references to the children, each node of the thousands generated is significantly smaller in size.

With that being said, each instance of a Node has the following attributes:

String state: This is where the state of the map is stored if we are to be in this node.

float cost: This is the cost needed to reach this node.

Node parent: This is a reference to the parent node, or in other words, the node that generated this node.

Action action: This is an ENUM that indicates the action taken by the parent node to generate this node

int depth: This indicates the depth of this node in the tree.

Other than the getters and setters, this class has other methods that help interact with this class.

## **heuristic1()/heuristic2():**

Methods that calculate the heuristic cost of this node.

## **generateNewNode(action, cost, depth):**

creates a new node based on the current node’s, state, attributes and the action passed down.

## **createState(previousState, action):**

returns a string depicting the new state that would result from doing such an action.

## **expand():**

returns an arraylist of nodes. Each new node represents a choice that the agent can follow through. In other words, actions that the agent cannot implement from this node are not generated, such as moving down when at the bottom of the grid or carrying on an empty node.

## **visitedBefore():**

determines whether or not this is a repeated state based on Ethan's position, number of agents carried on the truck, and the number of agents on the submarine.

## **canCarry():**

determines whether or not Ethan can carry an agent

## **getEthanX()/getEthanY():**

returns Ethan’s current x/y respectively.

## getAgentsOnSubmarine**/Truck/Dead():**

Does some string manipulation to return the number of agents on submarine/truck/dead.

## getPositionsArray():

Does some string manipulation to return an array of points of the agents’ locations on the map

## getHealthArray():

## Does some string manipulation to return an array of int of the agents’ healths.

## getIsCarriedArray():

Does some string manipulation to return an array of ints to determine whether or not this agent is carried.

## generateStateString():

takes state variables in their original form to transform them into a string that represents the state.

## stringfy():

takes an array and turns into into a string

## updateHealth():

updates the healths of the agents on the board

# **MissionImpossible Class:**

The *MissionImpossible* class extends the abstract class *SearchProblem.* It inherits and overrides the methods *solve, ids, dfs, bfs, isGoalState,* and *printSolution* from *SearchProblem.*

In addition to the inherited methods, the *MissionImpossible* class has *getNumberOfAgents, getSearchType, getWidth, getHeight, getSubmarinePosition, getGraphicalGrid, genGrid, ucs, greedy1, greedy2, astar1,* and *astar2.*

## **getNumberOfAgents:**

A method that returns the number of agents in the problem which is an integer value.

## **getSearchType:**

A method that returns the which search algorithm the MissionImpossible class uses to find the answer to the search problem. It returns a string.

## **getWidth:**

A method that returns the width of the problem grid, which is an integer.

## **getHeight:**

A method that returns the height of the problem grid, which is an integer.

## **getSubmarinePosition**:

A method that returns a point indicating the position of the submarine in the problem grid.

## **getGraphicalGrid:**

A method that returns the grid of the problem. It returns an object of type Grid. The Grid object is a class created to display the layout of the problem grid in 2D.

## **genGrid:**

A method that returns a string containing the layout of the grid of the problem. The string contains the width and height of the grid, Ethan’s initial position, the submarine’s position, all of the agents’ positions and healths, and the car capacity. All of the information in the string is generated at random such that every time the method is called a different string with different width, height, Ethan’s location, submarine’s location and all agents’ locations is generated.

## **Solve:**

A method that takes as parameters the search problem grid string, the strategy, and a boolean to determine whether to visualize the solution or not. The method splits the search problem grid string to extract the information about Ethan’s location, the submarine’s location, the agents’ locations and healths, and the car capacity. After extracting the required information from the grid string, it generates an initial state string and an initial node which will be the root of the search tree. The method then checks which strategy is chosen and whether the visualization boolean is set to true or not and accordingly it calls the method of the chosen strategy. Every strategy has two implemented methods to solve the search problem with them, a normal method that takes the node as a parameter and a second method that takes the visualization boolean in addition to the node as parameters. The difference between the two methods of every search strategy is when the visualization is set to true, a GUI appears to show the user the solution path rather than just printing the solution path as a string. The solve method also initializes the stack, the queue, and the priority queue used to store the nodes to be expanded as well as initializing the hashmap that stores the visited states to ensure that no state is visited twice.

## **Ids:**

A method that is called when the search strategy is iterative deepening search. There are two versions of this method, one that takes only the initial node and the depth of the node as parameters and the other takes the initial node, the depth of the node and visualization boolean as parameters. The ids method uses a stack to store the nodes to be expanded in it. While the stack is not empty, the method pops a node and checks if it is a goal state. If it is a goal state, the node is returned and if it is not a goal state, the method checks if the children of this node are still within the allowed depth. If they are, they are added to the stack. Every time the method pops a node, it checks if it was visited before using the method *visitedBefore()* in the Node class. If the stack is empty, then the method has failed to find a solution to the search problem within the allowed depth. The method is then called again after increasing the limit depth by 11. The same process is repeated until a solution is found within the allowed depth.

## **dfs:**

A method that is called when the search strategy is depth first search. There are two versions of this method, one that takes only the initial node as a parameter and the other takes the initial node and visualization boolean as parameters. Similar to ids, it uses a stack to store the nodes to be expanded in it. While the stack is not empty, the method pops a node and checks if it is a goal state. If it is a goal state, the node is returned and if it is not a goal state, the method adds the node’s children/expands the node and pushes the expanded nodes into the stack. Every time the method pops a node, it checks if it was visited before using the method *visitedBefore()* in the Node class. If the stack is empty, then the method has failed to find a solution to the search problem and returns null.

## **bfs:**

A method that is called when the search strategy is breadth first search. There are two versions of this method, one that takes only the initial node as a parameter and the other takes the initial node and visualization boolean as parameters. The method uses a queue to store nodes to be expanded. While the queue is not empty, the method dequeues a node from the queue and checks if it is a goal state. If it is a goal state, the node is returned and if it is not a goal state, the method expands the node’s children and enqueues them in the back of the queue. Every time the method dequeues a node, it checks if it was visited before using the method *visitedBefore()* in the Node class. If the queue is empty, then the method has failed to find a solution to the search problem and returns null.

## **ucs:**

A method that is called when the search strategy is uniform cost search. There are two versions of this method, one that takes only the initial node as a parameter and the other takes the initial node and visualization boolean as parameters. The method uses a priority queue to store the nodes to be expanded. The nodes are prioritized by their cost, the lower the cost, the higher the priority. The cost of the node is calculated by multiplying the number of dead agents who died in this node by 2000 in addition to the cost of the node’s parent. While the priority queue is not empty, the method dequeues a node from the priority queue and checks if it is a goal state. If it is a goal state, the node is returned and if it is not a goal state, the method expands the node’s children and enqueues them in the priority queue. Every time the method dequeues a node, it checks if it was visited before using the method *visitedBefore()* in the Node class. If the priority queue is empty, then the method has failed to find a solution to the search problem and returns null.

## **Greedy1:**

A method that is called when the search strategy is greedy using heuristic 1. There are two versions of this method, one that takes only the initial node as a parameter and the other takes the initial node and visualization boolean as parameters. The method uses a priority queue to store the nodes to be expanded. The nodes are prioritized by their cost, the lower the cost, the higher the priority. The cost of the node is calculated by the following equation cost(n) = h1(n) *(heuristic 1 is discussed in details below).* While the priority queue is not empty, the method dequeues a node from the priority queue and checks if it is a goal state. If it is a goal state, the node is returned and if it is not a goal state, the method expands the node’s children and enqueues them in the priority queue. Every time the method dequeues a node, it checks if it was visited before using the method *visitedBefore()* in the Node class. If the priority queue is empty, then the method has failed to find a solution to the search problem and returns null.

## **Greedy2:**

A method that is called when the search strategy is greedy using heuristic 2. There are two versions of this method, one that takes only the initial node as a parameter and the other takes the initial node and visualization boolean as parameters. The method uses a priority queue to store the nodes to be expanded. The nodes are prioritized by their cost, the lower the cost, the higher the priority. The cost of the node is calculated by the following equation cost(n) = h2(n) *(heuristic 2 is discussed in details below).* While the priority queue is not empty, the method dequeues a node from the priority queue and checks if it is a goal state. If it is a goal state, the node is returned and if it is not a goal state, the method expands the node’s children and enqueues them in the priority queue. Every time the method dequeues a node, it checks if it was visited before using the method *visitedBefore()* in the Node class. If the priority queue is empty, then the method has failed to find a solution to the search problem and returns null.

## **Astar1:**

A method that is called when the search strategy is astar using heuristic 1. There are two versions of this method, one that takes only the initial node as a parameter and the other takes the initial node and visualization boolean as parameters. The method uses a priority queue to store the nodes to be expanded. The nodes are prioritized by their cost, the lower the cost, the higher the priority. The cost of the node is calculated by the following equation cost(n) = h1(n) *(heuristic 1 is discussed in details below)* + g(n), where g(n) is the cost of the parent node. While the priority queue is not empty, the method dequeues a node from the priority queue and checks if it is a goal state. If it is a goal state, the node is returned and if it is not a goal state, the method expands the node’s children and enqueues them in the priority queue. Every time the method dequeues a node, it checks if it was visited before using the method *visitedBefore()* in the Node class. If the priority queue is empty, then the method has failed to find a solution to the search problem and returns null.

## **Astar2:**

A method that is called when the search strategy is astar using heuristic 2. There are two versions of this method, one that takes only the initial node as a parameter and the other takes the initial node and visualization boolean as parameters. The method uses a priority queue to store the nodes to be expanded. The nodes are prioritized by their cost, the lower the cost, the higher the priority. The cost of the node is calculated by the following equation cost(n) = h2(n) *(heuristic 2 is discussed in details below)* + g(n), where g(n) is the cost of the parent node. While the priority queue is not empty, the method dequeues a node from the priority queue and checks if it is a goal state. If it is a goal state, the node is returned and if it is not a goal state, the method expands the node’s children and enqueues them in the priority queue. Every time the method dequeues a node, it checks if it was visited before using the method *visitedBefore()* in the Node class. If the priority queue is empty, then the method has failed to find a solution to the search problem and returns null.

## **isGoalState:**

A method that takes as a parameter a node and checks whether this node is a goal state or not. The method returns a boolean value either true or false. A node is determined to be a goal state if Ethan's location in the node state is the same as the submarine’s location and if the number of agents on the submarine is equal to the number of total agents which means that Ethan has successfully carried all agents and dropped them on the submarine.

**Path-Cost Function:**

Each node created has its path cost. The path cost function is created to minimize the number of dead agents and the damage taken by each agent. The path cost accumulates over time since the node takes the path cost of its parent node in addition to its own path cost. The path cost function gives a large cost value for dead agents to make the search algorithm avoid taking this node. And it adds the total damage taken in this turn to make the search algorithm minimize the total damage taken for all agents. The optimal path for our search agent only of minimum damage and death, and does not depend on distance or number of steps taken.

**Heuristic 1 Function:**

The Heuristic function's main purpose is to give the agent a sense of what nodes should be avoided and try to stay near the agents and avoid paths that are far away from the agents. The heuristic function unlike the path cost does not accumulate its cost, each node has its heuristic cost. It calculates the distance between the current node and each alive and uncarried agent and for each agent, it is deemed to die because of the large distance it will give a large cost. The distance is calculated by the Manhattan Distance function[1]. For each move to reach the distance, the agent receives 2 damage points. So the distance is multiplied by 2 and added to the current health of the agent. If the health is greater than or equal to 100 then the agent is dead and the heuristic function adds a large cost value for the estimated dead agents.

**Heuristic 1 Admissible**:

The added value for the dead agents is equal to the added value for the dead agents in the path cost function. Since the path cost function adds value to each death in addition to the damage points taken by the alive agents, The heuristic function is only a part of the path cost so if it calculates that there is a dead agent the path cost for this route should also include the cost of the agent since the agent will die and for the cost of the damage taken by the agent before it dies. So the heuristic function will never overestimate the cost to reach any following state. Also for the goal state will have no uncarried agents so the heuristic cost will be equal to 0. Thus, making heuristic 1 admissible.

**Heuristic 2 Function:**

The heuristic 2 function has a similar concept to Heuristic 1 in that it also calculates the distance between the current node and each alive and uncarried agent. Also, if it is estimated that an agent will die it will give a large cost to avoid this node. Therefore, it also makes the agent avoid nodes that are far from the agents. However, the major difference between the 2 heuristics is the heuristic 2 calculates the distance by the Euclidean Distance function.[2] Since the Euclidean Distance is the direct distance from the current node to the agent, if the node and the agent are not on a horizontal or vertical distance it is almost guaranteed that the Euclidean Distance will be smaller than the actual distance traveled by the agent. Which gives the search agent an urgency to go and carry the agents that are about to die.

**Heuristic 2 Admissible**:

Similar to Heuristic 1, Heuristic 2 is also admissible. The path cost adds the cost of death and the damage point. Heuristic 2 only adds the cost of an estimated dead agent and neglects the damage points. Since the distance is calculated by the Euclidean Distance, the distance is already underestimated. And if the agent is already dead by this distance, it means if the search agent actually reached the agent, the agent will be dead and there might be additional damage points taken by other alive agents. Also, the death cost in this heuristic is the same as the path cost’s death cost. Also, for the goal node, there will be no uncarried agents so the heuristic cost will be 0. Thus making heuristic 2 admissible.

**Priority Queue:**

The priority queue is used to sort the node according to the node’s cost. The current node to expand is the node with the least cost among all nodes. The priority queue is used for the Uniform Cost search, Greedy 1&2, and A\* 1&2. So a node comparator class is implemented and is given to the priority queue to determine the cost that the queue will sort with. For example, given the node comparator with input ucs, the queue will only sort according to the path cost for each node. If the node comparator input is greedy1 then the queue will sort according to heuristic 1. The same is applied for greedy 2 the queue will sort according to the cost of the heuristic 2 cost of the nodes. If the node comparator input is astar1 then the queue will sort according to the path cost + heuristic 1 cost of the nodes. Also for astar2, the queue will sort according to the path cost + heuristic 2 costs of the nodes.

## **Grid Examples:**

### **Grid 1:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Agent 1  Health: 27  Position: 1, 0 |  |  | Agent 2  Health: 30  Position: 4, 0 |  |
|  |  |  |  |  |  |
|  |  |  |  | Submarine  Position: 4, 2 |  |
|  |  |  | Agent 3  Health: 60  Position: 3, 3 |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  | Ethan  Position: 1, 6  Carry Limit: 2 |  |  |  |  |
|  |  |  |  |  |  |
|  |  | Agent 4  Health: 90  Position: 2, 8 |  |  |  |

### 

### 

### 

### 

### **Grid 2:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Agent 1  Health: 30  Position: 0, 0 | Agent 2  Health: 50  Position: 1, 0 | Agent 3  Health: 20  Position: 2, 0 |  |  |  | Submarine |
|  |  |  |  |  |  |  |
| Ethan  Position: 0, 2  Carry Limit: 3 |  |  |  |  |  |  |
|  |  |  | Agent 4  Health: 10  Position: 3, 3 | Agent 5  Health: 20  Position: 4, 3 | Agent 6  Health: 80  Position: 5, 3 | Agent 7  Health: 84  Position: 6, 3 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

## **Performance Comparison:**

\*The graphs of the CPU usage spikes are always in this order grid 1| grid 2| grid 1| grid 2.

\*The RAM values are value that increased on the idle RAM value which is 606,644.

### **Breadth-First Search:**

1. Completeness:

The breadth-first search is complete as it always finds a solution, assuming there is one. In our case, the goal is to bring everyone to the submarine whether they are alive or dead. Meaning that there always will be a solution. Because even if all of the agents died the goal will be to bring them to the submarine. Of course, bringing them alive is better but what we mean is that even in the worst-case scenario the goal is achievable. After implementing the breadth-first search and testing it we found that it is in fact complete as it always provided us with a solution, like in grid 1 and grid 2 (mentioned above).

1. Optimality:

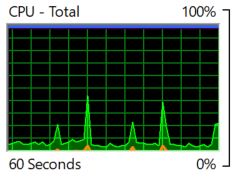
However, the breadth-first search is not optimal. Meaning achieving the best outcome is not always granted. Therefore, when using breadth-first search on grid 1 all 4 agents were taken to the submarine alive, however, in grid 2 of the 7 agents were taken to the submarine dead even though it was possible to take them alive.

1. RAM usage:

Grid 1: 136 KB

Grid 2: 136 KB

1. CPU utilization:



The CPU normally was at approximately 10% CPU usage of the Total CPU

The spikes that we see that are at 20% are when we implemented the

breadth-first search on grid 1. While grid 2’s CPU usage was at 40%. So we can

Say we have an average of 30%.

1. Expanded nodes:

Grid 1: 6544

Grid 2: 78277

### **Depth-First Search:**

1. Completeness:

The depth-first search is incomplete as it does not always find a solution, assuming there is one.

1. Optimality:

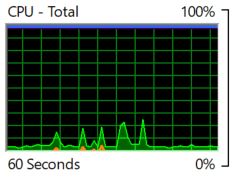
In addition, the depth-first search is not optimal. Therefore, when using depth-first search on grid 1 4 of the 4 agents were taken to the submarine dead, however, in grid 2 6 of the 7 agents were taken to the submarine dead even though it was possible to take them alive.

1. RAM usage:

Grid 1: 136 KB

Grid 2: 136 KB

1. CPU utilization:



The CPU normally was at approximately 10% CPU usage of the Total CPU

The spikes that we see that are at 20% are when we implemented the

depth-first search on grid 1 and 2. So we can Say we have an average of 20%.

1. Expanded nodes:

Grid 1: 800

Grid 2: 908

### **Iterative deepening search:**

1. Completeness:

The Iterative deepening search is complete as it does always find a solution, assuming there is one.

1. Optimality:

In addition, the Iterative deepening search is not optimal. Therefore, when

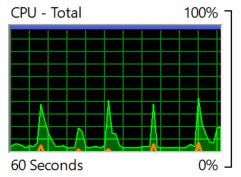
using Iterative deepening search on grid 1 3 of the 4 agents were taken to the submarine dead, however, in grid 2 6 of the 7 agents were taken to the submarine dead even though it was possible to take them alive.

1. RAM usage:

Grid 1: 116 KB

Grid 2: 248 KB

1. CPU utilization:



The CPU normally was at approximately 10% CPU usage of the Total CPU

The spikes that we see that are at 40% are when we implemented the

Iterative deepening search on grid 1 and 2. So we can Say we have an average of 40%.

1. Expanded nodes:

Grid 1: 30653

Grid 2: 290128

### **Uniform-cost search:**

1. Completeness:

The Uniform-cost search is complete as it does always find a solution, assuming there is one and the cost of the successor is more than the cost of the node. Meaning no negative cost paths. In our case, there are no negative cost paths and therefore, it's complete.

1. Optimality:

In addition, the Uniform-cost search is optimal assuming the cost of the successor is more than or equal the cost of the node. Therefore, when

using Uniform-cost search on grid 1 and 2 all of the agents were taken back

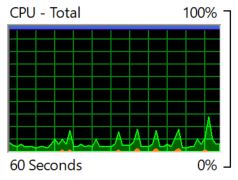
To the submarine alive.

1. RAM usage:

Grid 1: 132 KB

Grid 2: 136 KB

1. CPU utilization:



The CPU normally was at approximately 10% CPU usage of the Total

CPU The spikes that we see that are at almost 20% are when we implemented the Uniform-cost search on grid 1 and 2. So we can Say we have an average of 16%.

1. Expanded nodes:

Grid 1: 711

Grid 2: 2359

### **Greedy search:**

1. Completeness:

The Greedy search is not complete as it does not always find a solution, assuming there is one.

1. Optimality:

In addition, the Greedy search is not optimal. Therefore, when

using Greedy search on grid 1 Greedy 1 2 out of 4 agents were taken to the

Submarine dead. grid 2 Greedy 1 3 out of 6 agents were taken to the

Submarine dead. grid 1 Greedy 2 3 out of 4 agents were taken to the

Submarine dead. grid 2 Greedy 2 3 out of 6 agents were taken to the

Submarine dead.

1. RAM usage:
   1. Greedy 1:

Grid 1: 320 KB

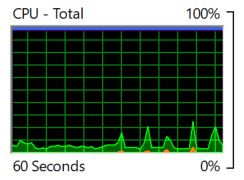
Grid 2: 136 KB

* 1. Greedy 2:

Grid 1: 158 KB

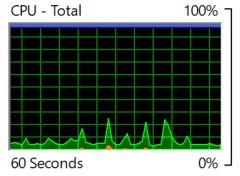
Grid 2: 156 KB

1. CPU utilization:



The spikes that we see that are at 10% are when we implemented the

Greedy 1 search on grid 1. While, the spikes that we see that are at 20% are when we implemented the Greedy 1 search on grid 2.



The spikes that we see that are at 10% are when we implemented the

Greedy 2 search on grid 1. While, the spikes that we see that are at 20% are when we implemented the Greedy 2 search on grid 2. So we can say the average is 15%.

1. Expanded nodes:
   1. Greedy 1:

Grid 1: 835

Grid 2: 1874

* 1. Greedy 2:

Grid 1: 623

Grid 2: 2283

### **A\* search:**

1. Completeness:

The A\* search is complete as it does not always find a solution, assuming there is one.

1. Optimality:

In addition, the A\* search is optimal. Therefore, when

using A\* search on grid 1 and 2 all of the agents were taken back

To the submarine alive.

1. RAM usage:
   1. A\* 1:

Grid 1: 136 KB

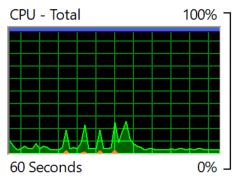
Grid 2: 136 KB

* 1. A\* 2:

Grid 1: 148 KB

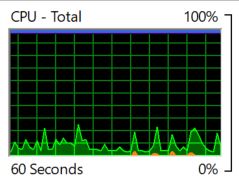
Grid 2: 135 KB

1. CPU utilization:



The spikes that we see that are at 20% are when we implemented the

A\* 1 search on grid 1 and 2.



The spikes that we see that are at 20% are when we implemented the

A\* 2 search on grid 1 and 2. So we can say the average is 20%.

1. Expanded nodes:
   1. A\* 1:

Grid 1: 585

Grid 2: 1653

* 1. A\* 2:

Grid 1: 584

Grid 2: 1881

### **Discussion:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Deaths | | RAM usage (+606,644 KB) | | Average CPU usage (%) | | Expanded nodes | |
|  | Grid 1 | Grid 2 | Grid 1 | Grid 2 | Grid 1 | Grid 2 | Grid 1 | Grid 2 |
| Breadth-first search | 0 | 2 | 136 | 136 | 30 | | 6544 | 78277 |
| Depth-first search | 4 | 6 | 136 | 136 | 20 | | 800 | 908 |
| Iterative deepening search | 3 | 6 | 116 | 248 | 40 | | 30653 | 290128 |
| Uniform-cost search | 0 | 0 | 132 | 136 | 16 | | 711 | 2359 |
| Greedy search 1 | 2 | 3 | 320 | 136 | 15 | | 835 | 1874 |
| Greedy search 2 | 3 | 3 | 158 | 156 | 15 | | 623 | 2283 |
| A\* search 1 | 0 | 0 | 136 | 136 | 20 | | 585 | 1653 |
| A\* search 2 | 0 | 0 | 148 | 135 | 20 | | 584 | 1881 |

\*Expanded nodes are not a measurement of memory as it depends how much are put at the same time.

As we can see from the results the depth-first search has the least nodes expanded and a small RAM usage and CPU usage in comparison. However, it has the worst optimality out of them all leading to 10 deaths. The breadth-first search had more CPU usage and more explored nodes than the depth-first search (same RAM usage) but, more optimal leading to 2 deaths. Also the iterative deepening search also had more RAM and CPU usage and explored nodes than depth-first and breadth-first search and that is probably because the iterative deepening search keeps on repeating the search from the start each time we increase the depth level. The iterative deepening search led to 9 deaths which is almost the same as the depth-first search. The Greedy search had the least CPU usage however, on average it had the highest RAM usage and a high number of expanded nodes and achieved a worse optimality than the breadth-first search. Lastly, the A\* and uniform cost search had close results. Almost same RAM usage, same deaths and almost same number of expanded nodes. However, the uniform cost search had a less CPU usage by 5%. That's probably due to the fact that in the A\* we calculate the g(n) and the h(n) while in the uniform cost we only calculate the g(n). In our case, the uniform cost achieved the same results like the A\* therefore, we can say that we can use the uniform cost search for this search problem and the A\* is not necessary to use as the uniform cost is capable.

## **Citations:**

[1] <http://www.improvedoutcomes.com/docs/WebSiteDocs/Clustering/Clustering_Parameters/Manhattan_Distance_Metric.htm#:~:text=The%20Manhattan%20distance%20function%20computes,differences%20of%20their%20corresponding%20components>.

[2] https://en.wikipedia.org/wiki/Euclidean\_distance