Artificial Intelligence Project 1 Report

This report includes some discussions of the implemented Abstract Data Types in this project along with some examples from the search algorithms' outputs. In addition, an evaluation between the search strategies is conducted by comparing the completeness, optimality, RAM usage and CPU utilization of each one of them.

1- Search Tree Node ADT Implementation

As defined by Russell and Norvig , the Node is defined by 5 attributes, so in our implementation the Node is a Class having 5 instance variables (currentState, parent, operator, cost, depth), with two additional instance variables holding the Heuristic costs for the Node (h1, h2). This way, whenever a node is to be created, the Node constructor is called.

2- Search Problem ADT implementation

Like the Node, the Search Problem is defined as an Abstract Class having 3 instance variables (setOperators, initialState, traversedStates) and 3 abstract methods (stateSpace, goalTest, pathCost). The Search problem is an abstract class, because no instances are created from it, however it is sub-classed (MissionImpossible Class).

3- Mission Impossible Implementation

The Mission Impossible is implemented as a SubClass of the Search Problem having various class attributes, like the grid coordinates, Truck Capacity, Ethan's position and the submarine's position, an array of IMF members (with their positions and healths), the number of expanded nodes and the limit depth of Iterative Deepening Search. Every time the constructor of the MissionImpossible is called new random values are assigned for the class attributes stated before or reverted to zero such as the case in the number of expanded nodes and the limit Depth. The values are assigned with respect to the constraints mentioned in the description grid are between 5*5 and 15*15, the number of generated IMF members is between 5 and 10. And no two objects (Ethan, Submarine, IMF member) are sharing the same cell. The cost of each step Ethan takes is calculated as the total damage occurred at that step for every team member and the cost of death is 1,000,000 to maximize the death cost compared to the damage.

4- The Main functions implemented in the MissionImpossible class are:

Overridden methods: (from the Search Problem Class)

-*State stateSpace(Operator operator, State parentState):*

transition function that returns the result State based on the parent State and the operator being applied on it.

-***int** pathCost(Node currentNode, Operator operator):*

Calculates the cost of a Node as a result of applying an operator to it and handles the cost of the death of an IMF member (+1000000).

-***boolean** goalTest(State currentState):*

checks if a state is a goal state or not.

===

-***int** getHeuristicManhatten(Node node):*

Returns the heuristic cost using the Manhattan distance

-***int** getHeuristicEuclidean(Node inp):*

Returns the heuristic cost using the Euclidean distance

***-boolean** checkEqualityOfMembers(ArrayList<Member> a1, ArrayList<Member> a2):*

Checks if two arrays of IMF members are the same or not.

***-boolean** checkRepeatedStates(State currentState):*

Checks if the State is a repeated one or not.

***-void** insertInTraversedStates(State currentState):*

Inserts the new states to the HashMap of the Repeated states.

-*ArrayList<Member> increaseDamage(ArrayList<Member> remaining):*

Handles the damage applied to all the remaining IMF members as a result of Ethan movement

- *String genGrid():*

generates all the random variables of the problem and returns a string to be passed to the solve() method

- **int** getPositionCost(ArrayList<Node> nodes, **int** cost)

- **int** getPositionH1(ArrayList<Node> nodes, **int** h1)

- **int** getPositionH2(ArrayList<Node> nodes, **int** h2)

- **int** getPositionAS1(ArrayList<Node> nodes, **int** cost)

- int getPositionAS2(ArrayList<Node> nodes, int cost)

Applies binary search to get the index at which the child node will be placed.

- String solve(String grid, String strategy, ***boolean*** *visualize):*

The core method, takes the problem string and returns a solution string, calls the transition function Qing_Func()if the goal is not reached.

- ArrayList<Node> Qing_Func(ArrayList<Node> nodes, QingFunction qing_fun):

Expands the nodes depending which search strategy is used.

-Node generalSearch(QingFunction qing_func):

keeps on expanding nodes(by calling Qing_Func) till a goal state is reached.

5- Implementation of the Search Strategies:

In our implementation, we have a queue implemented using an ArrayList called *nodes* that holds the nodes to be expanded. The program will loop on the queue until it is empty. The algorithm follows the steps below.

The check for the repeated states is done using a HashMap. The HashMap contains all the states and stores them with a key that is generated from the position of Ethan and the capacity of the truck. The values stored for each state is the array of remaining IMF members in this state. For handling the repeated states, the status of the IMF members was considered. We limit checking the status of every IMF member to either he was still bleeding or carried by Ethan. We excluded the damage of IMF members in the repeated states check as this leads to expanding a large set of new states that makes the Iterative Depth-first test cases - d5 and d6- to timeout.

1. Dequeue Node at position zero

2. Use the transition function stateSpace to get the next state for each operator in the set of operators.

3. Check if the state is repeated using *checkRepeatedState.*

4. If not repeated, create a new node with the corresponding new state and insert it into the queue according to the search strategy.

5. The above steps are repeated till no nodes are left in the queue.

1-Breadth First Search:

The child node is always appended at the end of the queue.

2-Depth First Search:

The child node is always appended at the beginning of the queue.

3- Uniform Cost Search:

The child node is placed at the index generated from the function *getPositionCost.* Nodes with higher <u>actual</u> cost are placed further down the queue.

4- Iterative deepening:

Iterative deepening follows the same steps with small modifications. The algorithm starts with limit depth = 0. In iterative deepening whenever the queue is empty the limitDepth is increased by 1 and the root of the search tree is placed at the beginning of the queue once more. If a childNode has a depth greater than the limitDepth, the childNode is discarded and not added to the queue. Nodes added when using iterative deepening are added at the beginning of the queue.

5- Greedy:

Greedy algorithms enqueue nodes, node at the index generated from the *getPoitionH1 and getPoitionH2* functions. Nodes with higher <u>heuristic</u> costs will be placed further down the queue.

6- A Star Search:

A-Star algorithms enqueue nodes, node at the index generated from the *getPoitionAS1 and getPoitionAS2* functions. Nodes with higher <u>heuristic</u> and <u>actual</u> costs will be placed further down the queue.

6- Implementation of the Heuristic functions

Two Heuristic functions were employed, one of them using the Manhattan distance and the other one using the Euclidean distance.

<u>*Heuristic Cost = 2 * min (IMF member distance) * size (remaining IMF members)*</u>

The Heuristic cost is equal to the minimum IMF member distance multiplied by 2 and multiplied by the number of remaining unsaved IMF members. The choice of factor 2 is because each step Ethan takes results in damage of 2 for each remaining IMF member. And In order for the Heuristic to be admissible, we multiply this 2*distance by the size of the remaining members. This way, we assume this minimal distance is the same for all the members, so the heuristic will always underestimate the cost. So, this Heuristic is admissible and dominant. In addition, we could have included the capacity of the truck in the equation; however, we wanted to remove some constraints so that the problem is relaxed.

7- Two running examples from your implementationand comparison

Regardless of the repeated nodes that the ID expands, the BFS expands the most number of nodes as it has a high space complexity.

UCS finds the solution with lowest costs but explores more nodes than DFS. DFS as expected expands a very low number of nodes because it has low space complexity but provides solutions with higher costs. We can see that both greedy algorithms expand the exact same number of nodes. And we can see the solution provided by AS1 and AS2 is the same as UCS. This is because these 3 algorithms provide the optimal solution with the lowest cost; however, AS1 and AS2 are able to find the solution with fewer node expansions.

In our implementation ID expands more nodes than BFS because we increase the number of expanded nodes every time we dequeue even though the children nodes may be eliminated because they are deeper than the maximum depth.

RAM and CPU utilization were highest in iterative deepening.

Grid5 = "5,5;2,1;1,0;1,3,4,2,4,1,3,1;54,31,39,98;2"

BFS

Solution:

```
down,down,right,carry,up,up,up,right,carry,left,left,left,drop,down,down,down,right,carry,up,carry,up,up,left,drop;1;981
```

number of nodes expanded: 981

RAM USAGE: 1.74%

CPU UTILIZATION: 7.3%

DFS

Solution:

```
up,up,right,right,down,down,down,down,left,left,carry,up,up,up,up,right,right,down,carry,up,left,left,down,left,drop,up,right,right,down,down,down,down,carry,up,up,up,up,left,left,down,drop,up,right,right,down,down,down,left,carry,up,up,left,drop;1;122
```

number of nodes expanded: 122

RAM USAGE: 0.96%

CPU UTILIZATION: 5.3%

UCS

Solution:

```
right,down,down,carry,left,carry,up,up,up,left,drop,right,right,right,carry,down,down,left,left,carry,up,up,left,
drop;1;237
```

number of nodes expanded: 237

RAM USAGE: 0.96%

CPU UTILIZATION: 5.2%

GR1:

Solution:

```
right,down,down,carry,left,carry,up,up,up,left,drop,right,right,right,carry,down,down,left,left,carry,up,up,left,
drop;1;90
```

number of nodes expanded: 90

RAM USAGE: 0.78%

CPU UTILIZATION: 4.9%

GR2:

```
right,down,down,carry,left,carry,up,up,up,left,drop,right,right,right,carry,down,down,left,left,carry,up,up,left,
drop;1;90
```

number of nodes expanded: 90

RAM USAGE: 0.78%

CPU UTILIZATION: 5.0%

ID:

Solution:

```
up,up,right,right,down,carry,up,left,left,down,down,down,carry,up,up,left,drop,up,right,right,down,down,do
wn,down,carry,left,carry,up,up,up,left,drop;1;13285
```

number of nodes expanded: 13285

RAM USAGE: 6.1%

CPU UTILIZATION: 9.8%

AS1:

Solution:

```
right,down,down,carry,left,carry,up,up,up,left,drop,right,right,right,carry,down,down,left,left,carry,up,up,left,
drop;1;182
```

number of nodes expanded: 182

RAM USAGE: 0.96%

CPU UTILIZATION: 7.3%

AS2:

Solution:

```
right,down,down,carry,left,carry,up,up,up,left,drop,right,right,right,carry,down,down,left,left,carry,up,up,left,
drop;1;181
```

number of nodes expanded: 181

RAM USAGE: 0.96%

CPU UTILIZATION: 7.2%

For Grid5, every Search strategy of our implementation in this running example finds a solution as there is at least one solution so our implementation for BF, DF, UC, ID, G, AS were complete.

The Breadth First, Depth First, and Iterative Deepening did not find the optimal solution as there was no favor to the total damage or the number of death(in the node expansion and enqueuing the nodes), so the three algorithms find the solution but it was not the optimal one as it can be concluded from the total cost in the Greedy strategy and the A Star (with minimal costs equal to 1000046)

Grid5:

AS => 1000046

GR => 1000046

BF => 1000058

DF => 1000122

ID => 1000112

Grid9 = "9,9;8,7;5,0;0,8,2,6,5,6,1,7,5,5,8,3,2,2,2,5,0,7;11,13,75,50,56,44,26,77,18;2"

BFS:

Solution:

up,up,up,up,up,up,up,up,right,carry,left,carry,down,down,down,down,down,left,left,left,left,left,left,left,drop,up,up,up,up,right,right,right,right,right,right,right,carry,down,left,carry,down,down,down,left,left,left,left,left,left,drop,up,up,up,right,right,right,right,right,carry,left,left,left,carry,down,down,down,left,left,drop,down,down,down,right,right,right,carry,up,up,up,left,left,left,drop,right,right,right,right,right,right,carry,left,carry,left,left,left,left,left,drop;6;123384

nodes expanded: 123384

RAM USAGE: 15%

CPU utilization: 8.5%

DFS:

Solution:
up,up,up,up,up,up,up,up,right,carry,down,down,down,down,down,down,down,down,left,left,up,up,up,up,up,up,up,up,left,left,down,down,down,down,down,down,down,down,left,left,up,up,up,up,up,up,up,up,left,left,down,down,down,down,down,drop,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,right,right,up,up,up,up,up,up,up,up,right,right,down,down,down,down,down,carry,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,left,left,up,left,up,left,up,up,up,up,up,up,left,left,down,down,down,down,down,down,down,down,right,carry,up,up,up,up,up,up,up,up,left,left,down,down,down,down,down,left,drop,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,right,right,up,up,up,up,up,up,up,up,right,right,down,down,carry,up,up,right,right,down,down,down,down,down,down,down,down,left,left,up,up,up,up,left,up,left,up,up,up,left,left,down,down,down,down,down,down,down,dow

```
n,left,left,up,up,up,drop,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,right,rig
ht,up,up,up,up,up,up,up,up,right,right,down,right,carry,up,left,left,down,down,down,down,down,down,dow
n,down,left,left,up,up,up,up,up,up,up,up,left,left,down,down,down,down,down,left,drop,up,up,up,up,up,rig
ht,right,down,down,down,down,down,down,down,down,right,right,up,up,up,up,up,up,up,up,right,right,right
,carry,down,down,down,down,down,down,down,down,left,left,up,up,up,up,up,up,up,up,left,left,down,down
,down,down,down,down,down,down,left,left,up,up,up,left,drop,up,up,up,up,up,right,right,down,down,down
,down,down,down,down,down,right,right,up,up,up,up,up,up,right,carry,up,up,right,right,down,down,down,
down,down,down,down,down,left,left,up,up,up,up,left,up,left,up,up,up,left,left,down,down,down,down,dow
n,left,drop,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,right,right,up,up,up,ri
ght,carry,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,left,left,up,left,up,left,u
p,up,up,up,up,up,left,left,down,down,down,down,down,left,drop,up,up,up,up,up,right,right,down,down,carr
y,up,up,right,right,down,down,down,down,down,down,down,down,left,left,up,up,up,up,left,left,down,drop;8
;854
```

nodes expanded: 854

RAM USAGE: 2.27%

CPU utilization: 5.5%

UCS:

```
up,up,left,up,carry,left,carry,left,left,left,left,left,drop,right,up,up,up,right,carry,down,down,down,left,left,dro
p,up,up,up,right,right,right,right,right,right,carry,up,up,right,right,carry,down,down,down,down,down,left,left
,left,left,left,left,left,left,drop,up,up,up,up,up,right,right,right,right,right,right,right,carry,down,carry,down,dow
n,down,down,left,left,left,left,left,left,left,drop,up,up,up,right,right,right,right,right,carry,down,down,down,do
wn,down,down,left,left,carry,up,up,up,left,left,left,drop;4;27438
```

Nodes Expanded: 27438

RAM USAGE: 11.8%

CPU utilization: 8.8%

GR1:

```
left,left,left,left,carry,up,up,up,right,right,carry,left,left,left,left,left,drop,up,up,up,right,right,carry,right,right,ri
ght,carry,down,down,down,left,left,left,left,left,drop,up,up,up,right,right,right,right,right,right,carry,up,right,c
arry,down,down,down,down,left,left,left,left,left,left,left,drop,up,up,up,up,up,right,right,right,right,right,right,r
ight,carry,right,carry,down,down,down,down,down,left,left,left,left,left,left,left,left,drop,right,right,right,right,r
ight,right,carry,left,left,left,left,left,left,drop;6;673
```

NODES EXPANDED: 673

RAM USAGE: 1.48%

CPU UTILIZATION: 4.4%

GR2:

```
left,left,left,left,carry,up,up,up,right,right,carry,left,left,left,left,left,drop,up,up,up,right,right,carry,right,right,ri
ght,carry,down,down,down,left,left,left,left,left,drop,up,up,up,right,right,right,right,right,right,carry,up,right,c
arry,down,down,down,down,left,left,left,left,left,left,left,drop,up,up,up,up,up,right,right,right,right,right,right,r
ight,carry,right,carry,down,down,down,down,down,left,left,left,left,left,left,left,left,drop,right,right,right,right,r
ight,right,carry,left,left,left,left,left,left,drop;6;673
```

NODES EXPANDED: 673

RAM USAGE: 1.48%

CPU UTILIZATION: 4.8%

AS1:

```
up,up,left,up,carry,left,carry,left,left,left,left,left,drop,up,up,up,right,right,carry,down,down,down,left,left,dro
p,up,up,up,right,right,right,right,right,right,carry,up,up,right,right,carry,down,down,down,down,down,left,left
,left,left,left,left,left,left,drop,up,up,up,right,right,right,right,right,carry,up,right,right,carry,down,down,down,d
own,left,left,left,left,left,left,left,drop,up,up,up,up,up,right,right,right,right,right,right,right,carry,down,down,d
own,down,down,down,down,down,left,left,left,left,carry,up,up,up,left,left,left,drop;4;22562
```

NODES EXPANDED: 22562

RAM USAGE: 6.7%

CPU UTILIZATION: 7.7%

ID:

```
up,up,up,up,up,up,up,up,right,carry,down,down,down,down,down,down,down,down,left,left,up,up,up,up,u
p,up,up,up,left,left,down,down,down,down,down,down,down,down,left,left,up,up,up,up,up,up,up,up,left,lef
t,down,down,down,down,down,drop,up,up,up,up,up,right,right,down,down,down,down,down,down,down,d
own,right,right,up,up,up,up,up,up,up,up,right,right,down,down,down,down,down,carry,up,up,up,up,up,right
,right,down,down,down,down,down,down,down,down,left,left,up,left,up,left,up,up,up,up,up,up,left,left,dow
n,down,down,down,down,down,down,down,right,carry,up,up,up,up,up,up,up,up,left,left,down,down,down,
down,down,left,drop,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,right,right,
up,up,up,up,up,up,up,up,right,right,down,down,carry,up,up,right,right,down,down,down,down,down,down,
down,down,left,left,up,up,up,up,left,up,left,up,up,up,left,left,down,down,down,down,down,down,down,dow
n,left,left,up,up,up,drop,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,right,rig
ht,up,up,up,up,up,up,up,up,right,right,down,right,carry,up,left,left,down,down,down,down,down,down,dow
n,down,left,left,up,up,up,up,up,up,up,up,left,left,down,down,down,down,down,left,drop,up,up,up,up,up,rig
ht,right,down,down,down,down,down,down,down,down,right,right,up,up,up,up,up,up,up,up,right,right,right
```

```
,carry,down,down,down,down,down,down,down,down,left,left,up,up,up,carry,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,left,left,up,left,up,left,up,up,up,up,up,up,left,left,down,down,down,down,down,left,drop,up,up,up,up,up,right,right,down,down,carry,up,up,right,right,down,down,right,carry,up,up,left,left,down,down,down,left,down,left,down,left,drop;8;24354791
```

nodes expanded: 24354791

RAM: 56.6%

CPU: 13.9%

AS2:

```
up,up,left,up,carry,left,carry,left,left,left,left,left,drop,up,up,up,right,right,carry,down,left,left,down,down,drop,up,up,up,right,right,right,right,right,right,carry,up,up,right,right,carry,down,down,down,down,down,left,left,left,left,left,left,left,left,drop,up,up,up,right,right,right,right,right,carry,up,right,right,carry,down,down,down,down,left,left,left,left,left,left,left,drop,up,up,up,up,up,right,right,right,right,right,right,right,carry,down,down,down,down,down,down,down,down,left,left,left,left,carry,up,up,up,left,left,left,drop;4;23666
```

NODES EXPANDED: 23666

RAM USAGE: 7.8%

CPU UTILIZATION: 7.2%

For Grid9, every Search strategy of our implementation in this running example finds a solution as there is at least one solution so our implementation for BF, DF, UC, ID, G, AS were complete.

The Breadth First, Depth First, and Iterative Deepening did not find the optimal solution as there was no favor to the total damage or the number of death(in the node expansion and enqueuing the nodes), so the three algorithms find the solution but it was not the optimal one as it can be concluded from the total cost in the A Star (with minimal cost equal to 4000413)

Grid9:

AS => 4000413

GR => 6000428

BF => 6000390

DF => 8000459

ID => 8000459