Team 35
Gasser Khaled 40-18970
Islam Nasr 40-4936
Kariman Hossam 40-1398
Marwan Karim 40-2763

AI Project 1 Report

25th of November 2020

1. We have two implementations of the search-tree node: the generic search tree node ADT and the mission impossible problem node which extends the generic node.

1.1 Search Tree Node ADT

The abstract search tree node is a class that implements the comparable interface. It has 7 instance variables:

- **State (String):** The state that this node corresponds to which has all the details needed for planning.
- **Parent (Node):** The parent node associated with this node.
- **Operator (String):** The operator (action) applied to generate this node.
- **Depth (int):** The depth (level) of this node in the search tree.
- **Path Cost (int):** Sum of the cost of the operators from the root.
- **Heuristic Cost (int):** An estimate of the path cost from this node to a goal node.
- **Priority (int)**: Used to sort the nodes based on the path cost, the heuristic cost, or the estimated cost of the cheapest solution through this node.

The abstract search tree node has getter methods for all the mentioned attributes. Nevertheless, we only have setters for the following attributes:

- **Heuristic Cost**
- **Priority**

Furthermore, this class contains an abstract method formulateNodeToString() which returns a String that consists of the following information:

- Ethan’s position.
- The IMFs states: 0 corresponds to an IMF not carried, 1 corresponds to a carried IMF, and 2 corresponds to an IMF dropped at the submarine.
- Carry (How many IMF members the truck carry at the moment).

1.2 Mission Impossible Node

The mission impossible node is a class that extends the generic node. It has 5 instance variables:

- **State (String):** The state that this node corresponds to which has all the details needed for planning.
- **Parent (Node):** The parent node associated with this node.
- **Operator (String):** The operator (action) applied to generate this node.
- **Depth (int):** The depth (level) of this node in the search tree.
- **Path Cost (int):** Sum of the cost of the operators from the root.

The problem search tree node has getter methods for all the mentioned attributes and no setter methods.

In addition, this class overrides the abstract method formulateToString() from the generic node class.

2. Search Problem ADT

The abstract search problem is a class which has the following instance variables:

- **Operators (String array):** Actions available to the agent.
- **Repeated States (Hashmap):** To keep track of the repeated states.
- **Expanded Nodes (int):** The number of nodes expanded (visited).
- **Initial State (Node):** The node where the search starts.

Moreover, it has the abstract methods:

- **transitionFunction(Node node, String operator):** It takes a node and an operator and returns the possible node generated from performing this action.
- **goalTest(Node node):** It takes a node and returns true if this node is a goal state, false otherwise.
- **returnPath(Node node):** It takes a node and returns the search problem solution which is a String containing all the sequence of actions from the initial state to this node.
- **getInitialState():** Returns a node which is the initial state of the problem.
- **getOperators():** Returns a String array that has all the operators available to the agent in the problem.
- **calculateFirstHeuristic(Node node):** It takes a node and returns the estimated cost of the cheapest path from the state at this node to a goal state.

- **calculateSecondHeuristic(Node node):** It takes a node and returns the estimated cost of the cheapest path from the state at this node to a goal state.

There is also the general search static method:

- **search(SearchProblem problem, QueuingFunction queuingFunction):** It takes as inputs the search problem and the queueing function which determines the order in which the nodes will be chosen for expansion. It returns a solution (String containing all the sequence of actions from the initial state to this node) or failure (Empty String).

3. Mission Impossible Problem

Mission impossible problem class is a class that extends search problem it has the following instance variables:

- **Grid (Grid Object):** An object representing the grid to perform the search on. It contains:
    - Rows
    - Columns
    - xEthan: initial x position of Ethan.
    - yEthan: initial y position of Ethan.
    - xSub: x position of the submarine.
    - ySub: y position of the submarine.
    - xPoss: Array containing the initial x positions of the IMFs.
    - yPoss: Array containing the initial y positions of the IMFs.
    - Damages: Array containing the initial damages of the IMFs.
    - C: The maximum number of IMF members that the truck can carry at a time.
- **Initial State (Node):** The node where the search starts.
- **Operators (String array):** It has all the operators available for the agent in the problem which are (Up, Down, Right,Left, Carry, Drop) in our problem.

It has the following static variable:

- **Visualize (boolean):** Variable which is set with the solve methods third parameter where it determines whether the path of Ethan will be visualized on the screen or not.

This class also overrides the abstract methods from the SearchProblem class. Moreover,it has the following static methods:

- **generateNumber(int min, int max):** It takes two numbers as an input and generates a random number between the given range.
- **genGrid():** Generates a random grid: the dimensions of the grid which should be between 5x5 and 15x15, the starting position of Ethan and the submarine, the number of IMFs which should be between 5 and 10 and their locations, a random starting damage between 1 and 99 for every IMF member, and finally the number of IMFs the truck can carry.
- **solve(String grid, String strategy, boolean visualize):** it uses search to try to formulate a winning plan. Grid is string separated by commas, and semicolons. Strategy is a string which contains the search strategy that will be applied which are:
    - BF for breadth-first search.
    - DF for depth-first search.
    - ID for iterative-deepening search.
    - UC for uniform-cost search.
    - GR_i for greedy search, where i ∈ {1,2} determines which heuristic is used.
    - AS_i for A* search, where i ∈ {1,2} determines which heuristic is used.
- **visualize(Node node):** This method takes a node and returns a string that gives a visual representation of the grid as it undergoes the different actions of the discovered path (solution).

4. Main Functions Implemented

- **solve(String grid, String strategy, boolean visualize):** This function is implemented in the Mission Impossible class. This function takes a grid, the solving strategy, and boolean called visualize as input parameters. The function then creates an initial node with the information from the given grid, sets its priority to 0, and then it either visualizes or not based on the boolean input. The function then creates an instance of a search problem of type MissionImpossible and passes the grid and the initial state to it. Consequently, the function then depending on the given strategy decides the Queuing function to be used to solve the problem then it calls the function search and passes the search problem of type MissionImpossible and the queuing function, which in turn returns a solution to this search problem. This solution is a String that contains the path to be taken to solve the problem.

- **search(SearchProblem problem, QueuingFunction queuingFuntion):** This function is a static function that is applied in the abstract class SearchProblem. Depending on the queuing function given to it creates a queue and then it enqueus the initial node. Then the initial node is dequeued and its children are then enqueued with different priorities depending on the queuing function. Each search strategy has a different function:
    1. **Breadth-First:** nodes closest to the root have a higher priority to be dequeued first.
    2. **Depth-First strategy:** nodes that are the furthest from the root have a higher priority to be dequeued first.
    3. **Iterative Deepening:** has the same logic as the Depth First but it keeps trying to find a solution with different depth limits until a solution is reached.
    4. **Uniform Cost Search:** nodes are dequeued based on the least cost which is calculated based on the damage inflicted by each node operation.
    5. **Greedy using the first heuristic function:** nodes with the least heuristic cost are dequeued first, and this cost is calculated using the first function.
    6. **Greedy using the second heuristic function:** nodes with the least heuristic cost are dequeued first, and this cost is calculated using the second heuristic function.
    7. **A* using the first heuristic:** this strategy calculates the priority of each node by adding the actual cost to reach each node plus their heuristic cost based on the first heuristic function.
    8. **A* using the second heuristic function:** this strategy calculates the priority of each node by adding the actual cost to reach each node plus their heuristic cost based on the second heuristic function.

- **transitionFunction(Node node, String operator):** This function is overridden in the MissionImpossible class. It takes a node and an operator and returns the node after applying the operation on the given node and changing all its variables accordingly. This is done by obtaining all the contents of the given node and then checking the operation to be applied on it. If the operation to be applied is the opposite of the operation applied in the previous node and the state of the IMFs is the same the function returns null as this operation is not allowed to happen to avoid repeated states. However, if the operation is allowed to occur the function calculates all the damage caused by performing this operation and updates the damage level of each IMF. Finally the Function updates the position of Ethan and the number of IMFs Ethan can carry (C) in case the operation to be applied is either Carry or Drop which can only be applied in specific situations, for instance carry can only be applied if Ethan is at the same position of an uncarried IMF, and Drop can only be applied if Ethan is at the submarine.

- **formulateNode():** This function is implemented in the Node class, when called on a node it returns the nodes information (Ethan's Position, Ethan's carry, the damage of the IMF states, the states of the IMFs) as a String which can be later split by ";" and ",".

- **returnPath(Node node):** This Function is implemented in the MissionImpossible class. It gets the input node and loops on all its parent nodes and pushes them into a stack to store them. In addition, for every node pushed it calls the visualize() function to demonstrate each node in the path. Finally the function returns the operations of all nodes in the stack as one string which represents the exact path taken to reach this node.

5. Search Algorithms

Inside the abstract Search Problem class, we have all the different search strategies implemented in the search method, where the method takes any instance of a search problem and a queuing function and returns a failure or a solution containing all the sequence of actions from the initial state to the goal state. In a Priority Queue called queue, we enqueue the initial state of the given problem, and then switch case on the queuing function which tells us which search strategy we are going to apply:

- **ENQUEUE_AT_END (Breadth-First Search):**

  We initialize an integer called priority with value 0 that gets incremented each time we add a node to the queue to make sure that the nodes will always be enqueued at the end. Subsequently, we iterate in a while loop which does not terminate unless the queue gets emptied or a goal is found. Inside the loop, we remove the node at the head of the queue and increment the number of expanded nodes. Then, we check if this node passes the goal test. If it does, we clear the repeatedStates HashMap and we call the returnPath function and give it this node to return a solution which is a String containing all the sequence of actions from the initial state to this node. On the other hand, if it does not pass the goal test, we iterate over the operators of the problem and apply all of them by calling the transitionFunction giving it this node and the current operator. If this method returns a node, we formulate this node to String to check whether this state was already visited before in our repeatedStates HashMap (to avoid repeated states). If the node was not found in the repeatedStates HashMap, we add it to the HashMap. Next, we set the priority of this node to the current priority integer to make sure that the nodes are added at the end of the queue in the same order (sorted from least to greatest), and then enqueue the node (which will be added at the end of the queue). Afterwards, we increment the priority integer. This loop continues until a node that passes the goal test is found, or the queue gets emptied resulting in a failure.

- **ENQUEUE_AT_FRONT (Depth-First Search):**

  We initialize an integer called priority with an integer maximum value that gets decremented each time we add a node to the queue to make sure that the nodes will always be enqueued at the front. Subsequently, we iterate in a while loop which does not terminate unless the queue gets emptied or a goal is found. Inside the loop, we remove the node at the head of the queue and increment the number of expanded nodes. Then, we check if this node passes the goal test. If it does, we clear the repeatedStates HashMap and we call the returnPath function and give it this node to return a solution which is a String containing all the sequence of actions from the initial state to this node. On the other hand, if it does not pass the goal test, we iterate over the operators of the problem and apply all of them by calling the transitionFunction giving it this node and the current operator. If this method returns a node, we formulate this node to String to check whether this state was already visited before in our repeatedStates HashMap (to avoid repeated states). If the node was not found in the repeatedStates HashMap, we add it to the HashMap. Next, we set the priority of this node to the current priority integer to make sure that the nodes are added at the beginning of the queue in the same order (sorted from least to greatest), and then enqueue the node (which will be added at the front of the queue). Afterwards, we decrement the priority integer. This loop continues until a node that passes the goal test is found, or the queue gets emptied resulting in a failure.

- **ENQUEUE_AT_FRONT_IDS (Iterative Deepening Search):**

  We iterate forever in a loop which does not terminate unless a goal is found. We initialize an integer called priority with an integer maximum value that gets decremented each time we add a node to the queue to make sure that the nodes will always be enqueued at the front. We also initialize an integer called currentDepth with a value 0 that gets incremented gradually until a goal is found. Subsequently, we iterate in another while loop which does not terminate unless the queue gets emptied. Inside this loop, we remove the node at the head of the queue and increment the number of expanded nodes. Then, we check if this node passes the goal test. If it does, we clear the repeatedStates HashMap and we call the returnPath function and give it this node to return a solution which is a String containing all the sequence of actions from the initial state to this node. On the other hand, if it does not pass the goal test, we iterate over the operators of the problem and apply all of them by calling the transitionFunction giving it this node and the current operator. If this method returns a node, we formulate this node to String to check whether this state was already visited before in our repeatedStates HashMap (to avoid repeated states). If the node was not found in the repeatedStates HashMap, we add it to the HashMap. Next, we set the priority of this node to the current priority integer to make sure that the nodes are added at the beginning of the queue in the same order (sorted from least to greatest), and then enqueue the node (which will be added at the front of the queue). Afterwards, we decrement the priority integer. If the queue gets emptied, we go back to the other while loop then increment the currentDepth by 10 (we know that we should increment it by 1 but due to the timeout we did this). After that, we add the initial state to our queue once again, and finally clear the repeatedStates HashMap. Because the queue is not empty, we go back to the while loop once again.

- **ORDERED_INSERT (Uniform Cost Search):**

  We iterate in a while loop which does not terminate unless the queue gets emptied or a goal is found. Inside the loop, we remove the node at the head of the queue and increment the number of expanded nodes. Then, we formulate this node to String to check whether this state was already visited before in our repeatedStates HashMap (to avoid repeated states). If the node was not found in the repeatedStates HashMap, we add it to the HashMap. Next, we check if this node passes the goal test. If it does, we clear the repeatedStates HashMap and we call the returnPath function and give it this node to return a solution which is a String containing all the sequence of actions from the initial state to this node. On the other hand, if it does not pass the goal test, we iterate over the operators of the problem and apply all of them by calling the transitionFunction giving it this node and the current operator. If this method returns a node, we set the priority of this node to its path cost to make sure that the nodes with lower path cost are added at the beginning of the queue (sorted from least to greatest), and then enqueue the node (which will be added to the queue according to its path cost). This loop continues until a node that passes the goal test is found, or the queue gets emptied resulting in a failure.

- **HEURISTIC_FN1 (Greedy Search using 1st heuristic):**

  We iterate in a while loop which does not terminate unless the queue gets emptied or a goal is found. Inside the loop, we remove the node at the head of the queue and increment the number of expanded nodes. Then, we formulate this node to String to check whether this state was already visited before in our repeatedStates HashMap (to avoid repeated states). If the node was not found in the repeatedStates HashMap, we add it to the HashMap. Next, we check if this node passes the goal test. If it does, we call the returnPath function and give it this node to return a solution which is a String containing all the sequence of actions from the initial state to this node. On the other hand, if it does not pass the goal test, we iterate over the operators of the problem and apply all of them by calling the transitionFunction giving it this node and the current operator. If this method returns a node, we calculate its heuristic cost by calling calculateFirstHeuristic function. Afterwards, we set the priority of this node to its calculated heuristic cost to make sure that the nodes with lower heuristic cost are added at the beginning of the queue (sorted from least to greatest), and then enqueue the node (which will be added to the queue according to its heuristic cost). This loop continues until a node that passes the goal test is found, or the queue gets emptied resulting in a failure.

- **HEURISTIC_FN2 (Greedy Search using 2nd heuristic):**

  We iterate in a while loop which does not terminate unless the queue gets emptied or a goal is found. Inside the loop, we remove the node at the head of the queue and increment the number of expanded nodes. Then, we formulate this node to String to check whether this state was already visited before in our repeatedStates HashMap (to avoid repeated states). If the node was not found in the repeatedStates HashMap, we add it to the HashMap. Next, we check if this node passes the goal test. If it does, we call the returnPath function and give it this node to return a solution which is a String containing all the sequence of actions from the initial state to this node. On the other hand, if it does not pass the goal test, we iterate over the operators of the problem and apply all of them by calling the transitionFunction giving it this node and the current operator. If this method returns a node, we calculate its heuristic cost by calling calculateSecondHeuristic function. Afterwards, we set the priority of this node to its calculated heuristic cost to make sure that the nodes with lower heuristic cost are added at the beginning of the queue (sorted from least to greatest), and then enqueue the node (which will be added to the queue according to its heuristic cost). This loop continues until a node that passes the goal test is found, or the queue gets emptied resulting in a failure.

- **EVALUATION_FN1 (A* Search using 1st admissible heuristic):**

  We iterate in a while loop which does not terminate unless the queue gets emptied or a goal is found. Inside the loop, we remove the node at the head of the queue and increment the number of expanded nodes. Then, we formulate this node to String to check whether this state was already visited before in our repeatedStates HashMap (to avoid repeated states). If the node was not found in the repeatedStates HashMap, we add it to the HashMap. Next, we check if this node passes the goal test. If it does, we call the returnPath function and give it this node to return a solution which is a String containing all the sequence of actions from the initial state to this node. On the other hand, if it does not pass the goal test, we iterate over the operators of the problem and apply all of them by calling the transitionFunction giving it this node and the current operator. If this method returns a node, we calculate its heuristic cost by calling calculateFirstHeuristic function. Afterwards, we set the priority of this node to its calculated heuristic cost added to its path cost to make sure that the nodes with lower path cost + heuristic cost are added at the beginning of the queue (sorted from least to greatest), and then enqueue the node (which will be added to the queue according to its cost). This loop continues until a node that passes the goal test is found, or the queue gets emptied resulting in a failure.

- **EVALUATION_FN2 (A* Search using 2nd admissible heuristic):**

  We iterate in a while loop which does not terminate unless the queue gets emptied or a goal is found. Inside the loop, we remove the node at the head of the queue and increment the number of expanded nodes. Then, we formulate this node to String to check whether this state was already visited before in our repeatedStates HashMap (to avoid repeated states). If the node was not found in the repeatedStates HashMap, we add it to the HashMap. Next, we check if this node passes the goal test. If it does, we call the returnPath function and give it this node to return a solution which is a String containing all the sequence of actions from the initial state to this node. On the other hand, if it does not pass the goal test, we iterate over the operators of the problem and apply all of them by calling the transitionFunction giving it this node and the current operator. If this method returns a node, we calculate its heuristic cost by calling calculateSecondHeuristic function. Afterwards, we set the priority of this node to its calculated heuristic cost added to its path cost to make sure that the nodes with lower path cost + heuristic cost are added at the beginning of the queue (sorted from least to greatest), and then enqueue the node (which will be added to the queue according to its cost). This loop continues until a node that passes the goal test is found, or the queue gets emptied resulting in a failure.

6. Heuristic Functions

The two heuristic functions that we used in our project for Greedy and A* search strategies are:

- **HEURISTIC 1**

  In our first heuristic function, we calculate the estimated heuristic cost by relaxing the constraints on our problem. This is done by calculating the distance from Ethan’s current position, to the most injured IMF agent, that would survive the damage Ethan would bring upon. As discussed earlier, the IMF agent takes 2 points of damage with every action that Ethan does, therefore, we calculate the damage the IMF agent will have to endure for Ethan to reach him by multiplying the Euclidean distance from Ethan to the IMF agent and multiplying that value by 2. However, that IMF agent has to not die till Ethan reaches him. The estimated cost to reach the goal is that damage that the IMF agent would have to endure due to Ethan’s actions to carry the IMF.

  This heuristic is admissible due to the fact that we have relaxed the problem significantly. The goal is assumed to be reaching a single IMF, that is severely damaged, instead of reaching all IMFs as well as ignoring the constraint related to the maximum capacity of the truck.:

- **HEURISTIC 2**

  In our second heuristic function, we calculate the estimated heuristic cost by relaxing the constraints on our problem. We calculate the heuristic by finding the closest IMF to Ethan and calculating the distance to reach that IMF agent. We then multiply this distance, which represents the number of actions Ethan has to perform, by 2 and then by the number of IMF agents, because all the agents will have to endure that damage too. The cost to reach the goal is therefore estimated to be the cost to reach and carry the closest IMF to Ethan (by distance).

  This heuristic is admissible because of the fact that we have ignored many constraints, including the penalty for deaths as well as carrying the nearest IMF only, instead of all IMFs. This approximated goal is significantly underestimated compared to the actual goal cost.

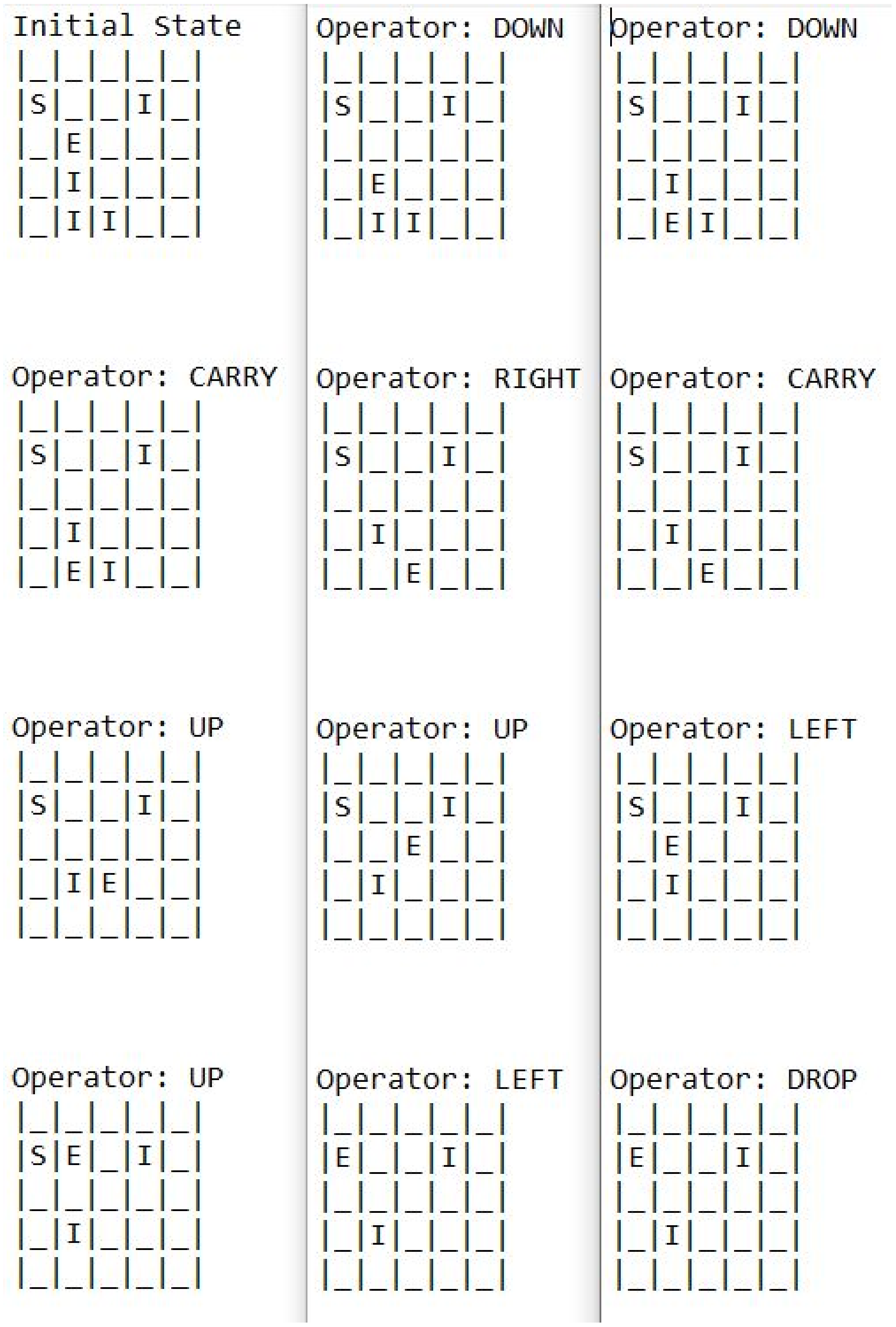
7. Examples

- **Example using visualize:**

  In the next few pictures, there is an example using the Solve method on a 5x5 grid with Uniform Cost search strategy. The result string returned from solve method was as follows:

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

  - down,down,carry,right,carry,up,up,left,up,left,drop,right,right,right,carry,left,down,left,down,carry,right,up,right,up,up,left,left,left,down,drop;1;82,39,43,100;271



```
Initial State
|_|_|_|_|_|
|S|_|_|I|_|
|_|E|_|_|_|
|_|I|_|_|_|
|_|I|I|_|_|
```

```
Operator: DOWN
|_|_|_|_|_|
|S|_|_|I|_|
|_|_|_|_|_|
|_|E|_|_|_|
|_|I|I|_|_|
```

```
Operator: DOWN
|_|_|_|_|_|
|S|_|_|I|_|
|_|_|_|_|_|
|_|I|_|_|_|
|_|E|I|_|_|
```

```
Operator: CARRY
|_|_|_|_|_|
|S|_|_|I|_|
|_|_|_|_|_|
|_|I|_|_|_|
|_|E|I|_|_|
```

```
Operator: RIGHT
|_|_|_|_|_|
|S|_|_|I|_|
|_|_|_|_|_|
|_|I|_|_|_|
|_|_|E|_|_|
```

```
Operator: CARRY
|_|_|_|_|_|
|S|_|_|I|_|
|_|_|_|_|_|
|_|I|_|_|_|
|_|_|E|_|_|
```

```
Operator: UP
|_|_|_|_|_|
|S|_|_|I|_|
|_|_|_|_|_|
|_|I|E|_|_|
|_|_|_|_|_|
```

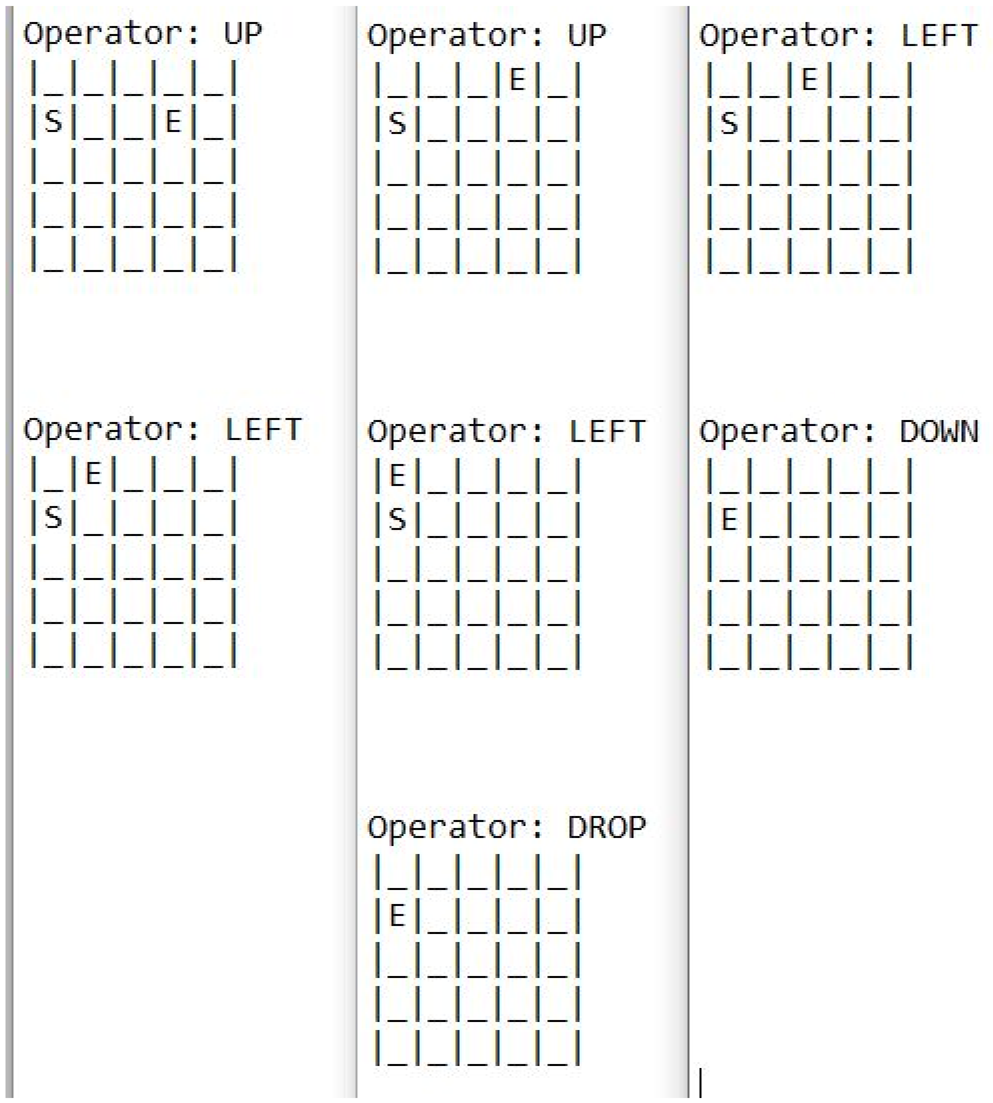
```
Operator: UP
|_|_|_|_|_|
|S|_|_|I|_|
|_|_|E|_|_|
|_|I|_|_|_|
|_|_|_|_|_|
```

```
Operator: LEFT
|_|_|_|_|_|
|S|_|_|I|_|
|_|E|_|_|_|
|_|I|_|_|_|
|_|_|_|_|_|
```

```
Operator: UP
|_|_|_|_|_|
|S|E|_|I|_|
|_|_|_|_|_|
|_|I|_|_|_|
|_|_|_|_|_|
```

```
Operator: LEFT
|_|_|_|_|_|
|E|_|_|I|_|
|_|_|_|_|_|
|_|I|_|_|_|
|_|_|_|_|_|
```

```
Operator: DROP
|_|_|_|_|_|
|E|_|_|I|_|
|_|_|_|_|_|
|_|I|_|_|_|
|_|_|_|_|_|
```



```
Operator: UP
|_|_|_|_|_|
|S|_|_|E|_|
|_|_|_|_|_|
|_|_|_|_|_|
|_|_|_|_|_|
```

```
Operator: UP
|_|_|_|E|_|
|S|_|_|_|_|
|_|_|_|_|_|
|_|_|_|_|_|
|_|_|_|_|_|
```

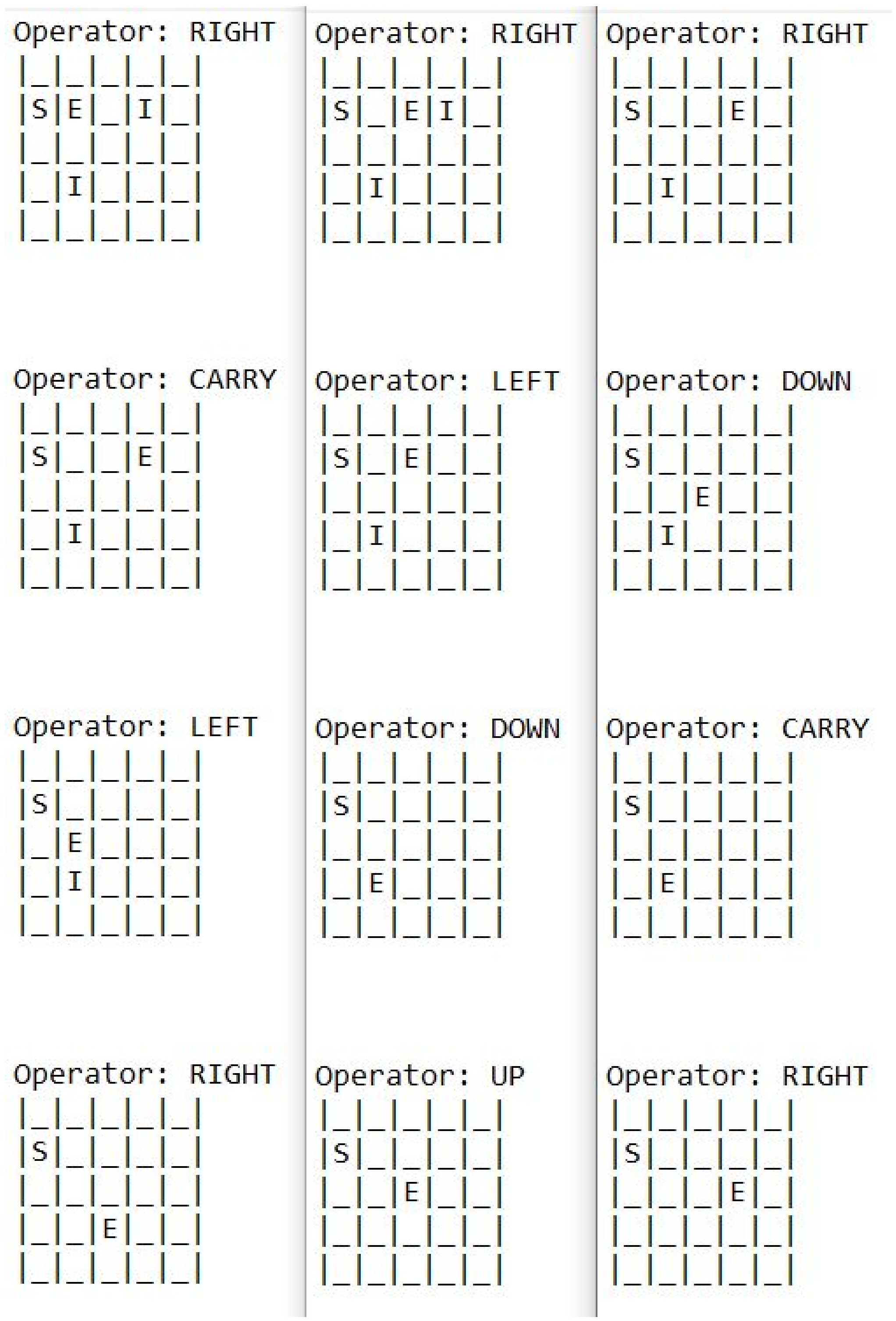
```
Operator: LEFT
|_|_|E|_|_|
|S|_|_|_|_|
|_|_|_|_|_|
|_|_|_|_|_|
|_|_|_|_|_|
```

```
Operator: LEFT
|_|E|_|_|_|
|S|_|_|_|_|
|_|_|_|_|_|
|_|_|_|_|_|
|_|_|_|_|_|
```

```
Operator: LEFT
|E|_|_|_|_|
|S|_|_|_|_|
|_|_|_|_|_|
|_|_|_|_|_|
|_|_|_|_|_|
```

```
Operator: DOWN
|_|_|_|_|_|
|E|_|_|_|_|
|_|_|_|_|_|
|_|_|_|_|_|
|_|_|_|_|_|
```

```
Operator: DROP
|_|_|_|_|_|
|E|_|_|_|_|
|_|_|_|_|_|
|_|_|_|_|_|
|_|_|_|_|_|
```



```
Operator: RIGHT
|_|_|_|_|_|
|S|E|_|I|_|
|_|_|_|_|_|
|_|I|_|_|_|
|_|_|_|_|_|
```

```
Operator: RIGHT
|_|_|_|_|_|
|S|_|E|I|_|
|_|_|_|_|_|
|_|I|_|_|_|
|_|_|_|_|_|
```

```
Operator: RIGHT
|_|_|_|_|_|
|S|_|_|E|_|
|_|_|_|_|_|
|_|I|_|_|_|
|_|_|_|_|_|
```

```
Operator: CARRY
|_|_|_|_|_|
|S|_|_|E|_|
|_|_|_|_|_|
|_|I|_|_|_|
|_|_|_|_|_|
```

```
Operator: LEFT
|_|_|_|_|_|
|S|_|E|_|_|
|_|_|_|_|_|
|_|I|_|_|_|
|_|_|_|_|_|
```

```
Operator: DOWN
|_|_|_|_|_|
|S|_|_|_|_|
|_|_|E|_|_|
|_|I|_|_|_|
|_|_|_|_|_|
```

```
Operator: LEFT
|_|_|_|_|_|
|S|_|_|_|_|
|_|E|_|_|_|
|_|I|_|_|_|
|_|_|_|_|_|
```

```
Operator: DOWN
|_|_|_|_|_|
|S|_|_|_|_|
|_|_|_|_|_|
|_|E|_|_|_|
|_|_|_|_|_|
```

```
Operator: CARRY
|_|_|_|_|_|
|S|_|_|_|_|
|_|_|_|_|_|
|_|E|_|_|_|
|_|_|_|_|_|
```

```
Operator: RIGHT
|_|_|_|_|_|
|S|_|_|_|_|
|_|_|_|_|_|
|_|_|E|_|_|
|_|_|_|_|_|
```

```
Operator: UP
|_|_|_|_|_|
|S|_|_|_|_|
|_|_|E|_|_|
|_|_|_|_|_|
|_|_|_|_|_|
```

```
Operator: RIGHT
|_|_|_|_|_|
|S|_|_|_|_|
|_|_|_|E|_|
|_|_|_|_|_|
|_|_|_|_|_|
```

- **Other Examples:**

  Input Grid:
  Grid A:

```
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
```

  Strategies:

  1- Depth First:

  up,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,down,right,right,up,up,up,up,up,up,up,up,up,right,right,down,down,down,carry,up,up,up,left,left,down,down,down,down,down,down,down,down,down,left,left,up,up,up,up,up,up,up,up,up,left,left,down,down,down,down,down,down,down,down,down,left,left,up,up,up,up,up,left,carry,up,up,up,up,right,right,down,down,down,down,down,down,down,down,down,right,right,up,up,up,up,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,down,right,right,up,up,up,up,up,up,up,up,right,carry,up,left,left,down,down,down,down,right,drop,up,up,up,up,left,left,down,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,down,left,carry,up,up,up,up,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,down,right,right,up,up,up,up,up,up,up,up,up,right,right,down,down,down,down,right,drop,up,up,up,up,left,left,down,down,down,down,down,down,down,down,down,left,left,up,up,up,up,up,up,up,carry,up,up,right,right,down,down,down,down,down,down,down,down,down,right,right,up,up,up,up,up,drop,up,up,up,up,left,left,down,down,down,down,down,down,down,down,down,left,left,up,up,up,up,up,up,carry,up,up,up,right,right,down,down,down,down,down,down,down,down,down,right,right,up,up,up,up,up,drop,up,up,up,up,left,left,down,down,down,down,down,down,down,down,down,left,left,up,up,up,carry,up,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,down,right,right,up,up,up,up,up,drop,up,up,up,up,left,left,down,carry,up,right,right,down,down,down,down,drop,up,up,up,up,left,left,left,carry,down,down,down,down,down,down,down,down,down,right,right,up,up,up,up,up,right,drop;8;100,100,100,83,100,100,100,100,100;980

  2- Breadth First

  right,carry,down,down,down,left,left,left,carry,up,up,up,up,up,left,carry,right,right,right,right,right,right,right,right,drop,up,up,up,up,left,left,left,carry,down,down,left,carry,down,carry,down,right,right,right,right,drop,up,up,up,left,left,carry,right,right,right

```
,carry,down,down,carry,down,left,drop;6;100,100,55,100,96,79,100,100,100;116225
7
```

3- Iterative Deepening

```
up,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,dow
n,right,right,up,up,up,up,up,up,up,up,up,right,right,down,carry,up,left,left,down,dow
n,down,down,down,down,down,down,down,left,left,up,up,up,up,up,up,up,up,up,lef
t,left,down,down,down,down,down,down,down,down,down,left,left,up,up,up,up,up
,left,carry,up,up,up,up,right,right,down,down,down,down,down,down,down,down,d
own,right,right,up,up,up,up,up,up,up,up,up,right,right,down,down,down,down,dow
n,down,down,down,down,right,right,up,up,up,up,up,drop,up,up,up,up,left,left,down
,down,down,down,down,down,down,down,down,left,left,up,up,up,up,up,up,up,up,
up,left,left,down,down,down,down,down,down,down,down,down,left,carry,up,up,u
p,up,up,up,up,up,up,right,right,down,down,down,down,down,down,right,carry,up,u
p,up,up,carry,up,up,right,right,down,down,down,down,down,down,down,down,do
wn,right,right,up,up,up,up,up,drop,up,up,up,up,left,left,down,carry,up,left,carry,dow
n,down,down,left,carry,right,right,up,right,up,right,right,down,down,down,left,drop,
up,right,carry,down,left,drop;9;100,100,100,100,100,100,100,100,100;7178828
```

4- Uniform Cost

```
right,carry,up,up,up,carry,up,carry,right,right,right,right,down,down,drop,right,up,ca
rry,left,up,up,left,left,carry,left,down,left,left,down,left,down,left,left,carry,right,right,r
ight,right,up,right,up,right,right,right,right,down,left,down,drop,up,left,left,down,left,
left,down,left,down,left,left,up,up,right,up,up,right,right,up,left,left,left,up,right,right,r
ight,right,carry,left,down,left,left,left,left,down,right,right,right,right,right,right,down,r
ight,right,down,drop,up,left,left,left,left,left,left,down,left,down,right,right,right,right,r
ight,down,down,down,down,left,left,left,left,up,left,left,up,up,up,up,up,up,up,right,ri
ght,up,right,right,right,right,right,down,right,up,right,down,carry,left,left,down,left,le
ft,left,down,right,right,down,right,down,left,down,left,left,down,left,left,down,down,l
eft,carry,left,up,up,right,up,up,right,right,down,right,right,right,right,right,right,up,up,
left,drop;3;100,63,91,51,96,13,100,81,100;4097
```

5- Greedy using heuristic 1

```
right,up,up,up,up,right,right,right,right,right,down,carry,up,left,left,left,up,carry,dow
n,left,left,left,left,left,down,down,left,down,down,down,down,right,right,up,right,righ
t,down,left,down,left,left,carry,up,up,up,up,up,right,right,right,right,right,right,right,d
rop,up,right,up,up,carry,down,down,down,left,left,left,up,up,left,left,carry,down,dow
n,right,right,right,right,drop,left,left,down,left,down,left,carry,up,up,right,right,right,u
p,up,up,up,left,left,carry,down,down,down,down,left,left,left,left,left,carry,right,right,
```

right,right,right,right,right,right,drop,left,left,left,left,up,carry,down,down,right,right,ri
ght,right,up,drop;7;100,100,100,39,100,100,100,69,100;32124

6- Greedy using heuristic 2

right,up,up,up,up,up,up,right,carry,down,down,left,carry,right,up,right,right,right,rig
ht,carry,down,down,down,left,drop,up,up,up,left,left,down,left,down,left,down,dow
n,down,carry,up,up,up,carry,up,up,right,right,carry,down,down,right,right,down,dro
p,up,left,left,left,left,down,left,left,left,down,down,down,down,down,carry,up,up,up,
up,up,right,right,right,up,right,right,right,right,right,carry,down,left,left,left,left,left,lef
t,left,left,left,carry,up,right,right,up,right,right,up,right,right,right,down,down,left,dow
n,right,right,drop;7;100,73,100,100,100,85,100,100,100;4495

7- A* using heuristic 1

right,carry,up,up,up,carry,up,carry,down,down,right,right,right,right,drop,right,up,ca
rry,up,up,left,left,left,carry,left,left,left,left,left,left,down,down,down,carry,down,dow
n,down,down,right,right,right,right,right,up,right,up,right,up,right,up,drop,down,dow
n,down,down,down,left,left,left,left,left,left,left,carry,up,up,up,up,up,right,right,right,
right,right,right,right,drop,up,up,up,right,carry,down,down,down,left,left,left,up,up,u
p,up,left,carry,down,down,down,right,down,right,right,drop;3;100,63,91,51,96,13,10
0,81,100;7493

8- A* using heuristic 2

right,carry,up,up,up,carry,up,carry,right,right,right,right,down,down,drop,up,right,ca
rry,up,up,left,left,left,carry,left,down,left,down,left,down,left,left,left,carry,down,dow
n,down,right,right,right,up,up,right,up,up,up,up,right,right,right,right,down,down,do
wn,drop,up,up,up,right,carry,up,left,left,left,left,carry,down,down,down,down,down,
down,down,down,down,left,left,left,left,carry,up,right,right,right,right,up,up,up,up,ri
ght,right,right,drop;3;100,63,91,51,96,13,100,81,100;4239

Input Grid:
Grid B:

```
12,12;7,7;10,6;0,4,2,2,1,3,8,2,4,2,9,3;95,4,68,2,94,91;5
```

Strategies:

1- Depth First

up,up,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,d
own,down,down,left,left,up,up,left,up,left,up,up,up,up,up,up,up,up,left,left,down,do
wn,down,down,down,down,down,down,down,carry,up,up,up,up,up,up,up,up,up,rig

ht,right,down,down,down,down,down,down,down,down,down,down,right,drop,up,up,up,up,up,up,up,up,up,up,left,left,down,down,down,down,down,down,down,down,down,down,down,left,left,up,up,up,up,up,up,up,up,up,carry,up,up,right,right,down,down,down,down,down,down,down,down,down,down,down,right,right,up,drop,up,up,up,up,up,up,up,up,up,up,left,left,down,down,down,down,down,down,down,down,down,down,down,left,left,up,up,up,up,up,up,up,carry,up,up,up,up,right,right,down,down,down,down,down,down,down,down,down,down,down,right,right,up,drop,up,up,up,up,up,up,up,up,up,up,left,left,down,down,down,down,down,down,down,down,down,down,down,left,left,up,up,up,carry,up,up,up,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,down,down,down,right,right,up,drop,up,up,up,up,up,up,up,up,up,up,left,left,down,left,carry,up,right,right,down,down,down,down,down,down,down,down,down,down,right,drop,up,up,up,up,up,up,up,up,up,up,left,left,carry,down,down,down,down,down,down,down,down,down,down,down,right,right,up,drop;6;100,100,100,100,100,100;1246

2- Breadth First

up,up,up,up,up,up,up,left,left,left,carry,down,left,carry,down,left,carry,down,down,carry,down,down,down,down,carry,down,down,right,right,right,right,drop,up,left,left,left,carry,down,right,right,right,drop;3;100,36,94,50,100,100;36055

3- Iterative Deepening

up,up,up,up,up,up,up,right,right,down,down,down,down,down,down,down,down,down,down,down,left,left,up,up,left,up,left,up,up,up,up,up,up,up,up,left,left,down,down,down,down,left,carry,up,up,up,up,right,right,carry,down,down,down,down,down,down,down,down,down,down,down,left,left,up,up,up,up,up,up,up,up,up,carry,up,right,carry,up,right,right,down,down,down,down,down,down,down,down,down,down,right,drop,up,up,left,left,down,left,carry,up,left,carry,down,down,down,right,right,up,right,right,drop;6;100,100,100,100,100,100;84556

4- Uniform Cost

left,left,down,left,left,left,carry,up,up,up,up,up,up,up,right,carry,down,left,carry,right,right,down,down,down,right,right,down,right,up,right,right,down,left,down,right,right,down,left,left,left,down,left,left,left,down,left,left,down,right,right,right,right,up,drop,up,left,left,left,carry,down,right,right,right,right,right,right,right,right,up,up,left,left,up,left,left,up,right,right,right,up,left,left,left,left,up,left,left,left,up,left,left,up,left,up,right,right,right,right,right,right,right,right,right,right,right,up,left,left,left,left,left,left,left,carry,left,left,left,down,down,left,down,right,right,right,down,left,carry,left,left,down

,down,right,right,right,right,right,right,right,right,right,right,down,left,down,left,down,left,left,left,down,right,drop;3;100,40,98,14,100,100;1201

5- Greedy using heuristic 1

up,up,up,up,up,up,left,left,left,up,carry,down,left,down,down,down,left,carry,up,up,carry,down,down,down,down,down,down,carry,down,down,right,right,right,right,drop,left,up,left,left,carry,down,right,right,right,up,up,up,up,up,up,up,up,left,left,up,left,carry,down,down,down,down,right,right,right,right,right,down,down,right,right,down,left,left,left,down,left,down,drop;4;100,44,100,56,100,100;878

6- Greedy using heuristic 2

down,down,left,left,left,left,carry,up,left,up,up,up,up,up,up,up,right,up,right,carry,down,left,carry,down,left,carry,down,down,carry,down,down,down,down,right,right,right,right,down,down,drop,up,up,left,left,left,left,carry,down,down,down,right,right,right,up,right,drop;4;100,54,100,94,100,100;306

7- A* using heuristic 1

left,left,left,left,left,down,carry,up,up,up,up,up,up,up,right,carry,down,left,carry,up,up,right,right,carry,down,down,down,down,down,down,down,down,down,down,right,right,drop,up,up,left,up,up,up,left,left,left,up,carry,down,down,down,down,down,right,carry,right,right,right,down,drop;3;100,40,98,14,100,100;1234

8- A* using heuristic 2

down,left,left,left,left,left,carry,up,up,up,up,up,up,right,up,carry,down,left,carry,down,down,carry,up,up,up,up,right,right,carry,down,down,down,down,down,down,down,down,down,right,right,down,drop,up,left,left,left,carry,down,down,right,right,right,up,drop;3;100,40,98,14,100,100;886

8. Performance Comparison

In this section, we will compare the performance of the search strategies in terms of completeness, optimality, RAM usage, CPU utilization, and the number of expanded nodes on the grids from the examples above.

- **Grid A:**

```
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
```

1. **Completeness:**

    - Breadth First

      BF is a complete search strategy. All the nodes are expanded level by level. It first expands all the nodes at the first level in the search tree, then expands all the nodes of the second level and so on until it reaches the goal. This strategy works in a FIFO (First In First Out) manner, where a node is expanded and then its children are added to the queue; thus, the search eventually reaches a goal state at some level.

    - Depth First

      In general, DF is an incomplete search strategy. The expansion starts from the root node and goes all the way up to the deepest unexpanded node in search of the goal. This way, it can get stuck in an infinite path. However, in our problem, we already eliminated the repeated states; therefore, it is guaranteed to reach a solution if it exists.

    - Iterative Deepening

      ID is a complete search strategy. It is basically doing DF in a BF manner because it restricts DF from going beyond a certain depth. Hence, if there is a solution, it is guaranteed to be found.

    - Uniform Cost

      UC is a complete search strategy. It expands the nodes with low cost first. Since we handle the repeated states in our problem, then if there is a solution, it is guaranteed to be found.

    - Greedy 1 & 2

      Greedy is a complete search strategy. It expands the nodes according to a heuristic function where nodes having a minimal heuristic value are expanded first. It also guarantees to find a solution at some level as we traverse a level at a time.

    - A* 1 & 2

      A* is a complete search strategy. It joins UC and Greedy together. It expands the nodes with less path cost from the root to the current node + the current node heuristic cost first. Therefore, it guarantees to find the optimal solution if it exists.

2. **Optimality:**

    - Breadth First

      BF is not an optimal search strategy in general. However, in our problem it can be considered optimal as all the nodes have the same cost because we do not take the path cost into consideration. Therefore, it returns the first solution it finds.

    - Depth First

      DF is not an optimal search strategy. It expands deeper nodes first, so it may find a non-optimal solution first.

    - Iterative Deepening

      ID is an optimal search strategy provided that the path cost is a non-decreasing function in the depth of the node. In our problem, we do not take into consideration the path cost and we will not get stuck in an infinite path, so it is optimal.

    - Uniform Cost

      UC is an optimal search strategy. Since the cost of a node is always less than or equal to its successors and the nodes are expanded according to the lowest path cost first, it is an optimal strategy.

    - Greedy 1 & 2

      Greedy search is not an optimal search strategy in general because it is not guaranteed to find the global lowest cost solution. It only finds a local optimal solution.

    - A* 1 & 2

      A* is an optimal search strategy as long as the heuristic is admissible which means that it never overestimates the cost of the path to the goal from any given node. Since we expand the nodes with the lowest path cost + heuristic cost first, we will always find the optimal solution.

3. **RAM usage and CPU utilization:**

Search Strategy	RAM Usage	CPU Utilization
Breadth First	1532 MB	22.1 %
Depth First	0.6 MB	2.1 %
Iterative Deepening	1448.2 MB	9.8 %
Uniform Cost	1.5 MB	8.2 %
Greedy 1	201 MB	3.5 %
Greedy 2	60 MB	6 %
A* 1	90.3 MB	7.1 %
A* 2	26.6 MB	4.3 %

- Breadth First

  BF has a considerably large RAM usage due to the fact that it has to keep track of every single node object, from the start of the search (root) till the very end of the search (goal). It expands nodes in a breadth, or horizontal fashion, meaning it will expand at most 6 nodes, from every given node. This creates an exponential increase in the number of nodes, which take up an exponentially large sum of memory. The CPU utilisation is also merited due to the fact that it has to process and store a huge number of nodes, which is very expensive.

- Depth First

  DF has a small RAM usage due to the fact that it expands the nodes in a zig-zag fashion, meaning it will move over the grid in all "up" actions for example, unless it's not possible to do any more "up", then it'll switch to the next possible move, which would be "down" for example. This will make the agent move across the whole grip in a looping manner, which ensures a minimal number of expanded nodes, which in turn does not consume a lot of memory. It also ensures finding a solution considerably fast, therefore the CPU utilisation is also minimal.

- Iterative Deepening

ID is a very expensive search strategy, this is due to the fact that it is implemented using multiple Depth First iterations, where the search tree is re-constructed from the root, up to a certain depth, n times, where n is the depth at which the goal is. ID follows the depth first strategy, up to a certain depth, then starts traversing the tree towards the right. This would make it store a considerably large amount of nodes, similar to BF. DF expands much more nodes compared to BF because many nodes, which are at the top of the tree, get expanded multiple times (sometimes n times) by the strategy, however, DF takes a little less memory than BF because we increment the depth limit by 10 instead of 1, to make the strategy perform a little faster. The CPU usage is also considerable, because of how many times the HashMap that contains the repeated states gets cleared, as well as the repeated expensive iterations.

- Uniform Cost

  UC expands the nodes based on their cost from the root, which forces the agent to expand less nodes and to order them based on the cost from the goal. This eventually leads to sensible node expansions, therefore the RAM usage is considerably lower than BF and ID. However, the CPU usage is not low, because of all the calculations that have to be made using the path cost function.

- Greedy 1

  GR1 is using the first heuristic function, which estimates considerably lower than actual cost values. This makes the agent enqueue and expand nodes that are less likely to be the goal, in a very greedy way. This produces a large number of node expansions, which takes up a big chunk of memory. Processing the heuristic cost function also takes CPU time, increasing the utilisation compared to DF (even though the heuristic function is calculated in O(number of remaining agents) time at worst), however the utilisation is still less than BF and ID for example, due to the very huge difference of expanded nodes.

- Greedy 2

GR2 is using the second heuristic function, which also estimates lower than actual cost values, however these estimates are a little closer to the actual cost compared to GR1 heuristic. This inturn makes the agent expand lesser nodes and eventually reach the goal earlier in the expansion process. However, this comes at the cost of the more expensive heuristic calculation function, which takes more time and processing power compared to the simpler first heuristic.

- A* 1

  A* 1 is very similar to GR1, however the nodes are expanded using a better estimation compared to using a greedy, heuristic based approach. This causes a fewer number of nodes to be expanded, which in turn uses less memory compared to GR1, however the memory used is still greater than A* 2 because the first heuristic estimates lower values than the second heuristic. However, this also comes at the cost of processing more, by calculating more values (the addition of g(n) and h(n) every time).

- A* 2

  A* 2 uses less memory compared to A* 1 because of using the second heuristic function which estimates closer to actual costs, which eventually leads to a lower number of ineffective expansions (expansions that don’t reach the goal). It also uses better CPU compared to A* 1 because of the lower number of expansions as well as the lower number of calculations.

4. **Number of expanded nodes:**

- Breadth First

  The number of expanded nodes in the first grid using BF strategy was 1162257, this number was high as expected as the agent goes to level by level solutions and by each level he passes a maximum of 6 nodes for each parent which is relevant to 6^n where n represents the depth of the solution. By handling repeated and unwanted states the 5^n is a rare case where it is usually a 4^n or less iterated nodes unless cases where we have an IMF to carry then we have 5^n and if we can drop the IMFs it is 5^n, of course we cannot carry and drop at the same time so despite that we have 6 operators we can do a maximum of 5 operators at the same node.

- Depth First

  The number of expanded nodes in the first grid using DF strategy was 980, this number was somewhat low which is what we expected as we handled repeating states so that means whenever we are searching deeper into the tree (further from the root) we are going towards the solution so even though our agent would take a maximum of n*m (where n is the width and m is the height) steps until he reaches a repeated state but it would eventually lead him to a solution.

- Iterative Deepening

  The number of expanded nodes in the first grid using ID strategy was 7178828, this number was higher than breadth first as expected, this is because it does depth first at different depth limit so it will iterate the same nodes multiple of times until it reaches a depth limit greater than or equal to a solution depth.

- Uniform Cost

  The number of expanded nodes in the first grid using UC strategy was 4097, this number was greater than depth first but as expected it is less than BF and ID with a huge difference this is due to expanding less nodes than because it doesn't expand nodes except if it have the lowest path cost compared to all other nodes in the queue.

- Greedy 1

  The number of expanded nodes in the second grid using GR1 strategy was 4495 which is somehow large compared to uniform cost, it makes sense indeed because it expands more nodes as more clashes happen in heuristic costs between nodes with some nodes having same heuristic costs so in turn more nodes are expanded than UC but of course it expands less nodes than ID and BF.

- Greedy 2

  The number of expanded nodes in the first grid using GR2 strategy was 4495 which is larger than uniform cost by small difference. Compared to greedy1 it extends less nodes, this reflects that our second heuristic function is a better heuristic function than our first function.

- A* 1

  The number of expanded nodes in the first grid using A* 1 strategy was 7493, as we expected it must be a value less than greedy function1 and higher than the uniform cost because it is expanding greater nodes because of the same priority nodes due to the addition of the uniform cost with heuristic cost which may lead to some costs being the same but at the end it will be less than the heuristic alone as clashes will occur.

- A* 2

  The number of expanded nodes in the first grid using A8-2 strategy was 4239, as we expected it is less than greedy function 2 and higher than uniform cost, as we explained for A* 1. We also expected that it will be less than A* 1 because the heuristic used in A* 2 is better (as we mentioned before) than the heuristic used in A* 1.

- **Grid B:**

```
12,12;7,7;10,6;0,4,2,2,1,3,8,2,4,2,9,3;95,4,68,2,94,91;5
```

1. **Completeness:**

   - Breadth First

     BF is a complete search strategy. All the nodes are expanded level by level. It first expands all the nodes at the first level in the search tree, then expands all the nodes of the second level and so on until it reaches the goal. This strategy works in a FIFO (First In First Out) manner, where a node is expanded and then its children are added to the queue; thus, the search eventually reaches a goal state at some level.

- Depth First

  In general, DF is an incomplete search strategy. The expansion starts from the root node and goes all the way up to the deepest unexpanded node in search of the goal. This way, it can get stuck in an infinite path. However, in our problem, we already eliminated the repeated states; therefore, it is guaranteed to reach a solution if it exists.

- Iterative Deepening

  ID is a complete search strategy. It is basically doing DF in a BF manner because it restricts DF from going beyond a certain depth. Hence, if there is a solution, it is guaranteed to be found.

- Uniform Cost

  UC is a complete search strategy. It expands the nodes with low cost first. Since we handle the repeated states in our problem, then if there is a solution, it is guaranteed to be found.

- Greedy 1 & 2

  Greedy is a complete search strategy. It expands the nodes according to a heuristic function where nodes having a minimal heuristic value are expanded first. It also guarantees to find a solution at some level as we traverse a level at a time.

- A* 1 & 2

  A* is a complete search strategy. It joins UC and Greedy together. It expands the nodes with less path cost from the root to the current node + the current node heuristic cost first. Therefore, it guarantees to find the optimal solution if it exists.

2. **Optimality:**

   - Breadth First

BF is not an optimal search strategy in general. However, in our problem it can be considered optimal as all the nodes have the same cost because we do not take the path cost into consideration. Therefore, it returns the first solution it finds.

- Depth First

  DF is not an optimal search strategy. It expands deeper nodes first, so it may find a non-optimal solution first.

- Iterative Deepening

  ID is an optimal search strategy provided that the path cost is a non-decreasing function in the depth of the node. In our problem, we do not take into consideration the path cost and we will not get stuck in an infinite path, so it is optimal.

- Uniform Cost

  UC is an optimal search strategy. Since the cost of a node is always less than or equal to its successors and the nodes are expanded according to the lowest path cost first, it is an optimal strategy.

- Greedy 1 & 2

  Greedy search is not an optimal search strategy in general because it is not guaranteed to find the global lowest cost solution. It only finds a local optimal solution.

- A* 1 & 2

  A* is an optimal search strategy as long as the heuristic is admissible which means that it never overestimates the cost of the path to the goal from any given node. Since we expand the nodes with the lowest path cost + heuristic cost first, we will always find the optimal solution.

3. **RAM usage and CPU utilization:**

Search Strategy	RAM Usage	CPU Utilization
Breadth First	36.1 MB	3.5 %

Depth First	0.9 MB	2.8 %
Iterative Deepening	38.3 MB	7.2 %
Uniform Cost	1.3 MB	2.7 %
Greedy 1	42.2 MB	1.5 %
Greedy 2	37.2 MB	2.1 %
A* 1	12.5 MB	4.4 %
A* 2	8.6 MB	2.7 %

- Breadth First

  BF has a considerably large amount of RAM usage because it expands nodes in an ever expanding fashion, which also uses from the CPU. Moreover, the number of expanded nodes is much less than the previous example, even though the grid size here is larger.

- Depth First

  DF has a small RAM and CPU utilisation for the same reasons as the previous example (searching the grid in a looping manner, which forces a small number of expansions).

- Iterative Deepening

  ID is a very expensive search strategy, this is due to the fact that it is implemented using multiple Depth First iterations, where the search tree is re-constructed from the root, up to a certain depth, n times, where n is the depth at which the goal is. The same reasons from the first example explanation apply here too, however the contrast between BF, DF and ID RAM usage is not as stark, which is probably an anomaly in the measurement application even though the CPU usage is consistent.

- Uniform Cost

  UC expands the nodes based on their cost from the root, which forces the agent to expand less nodes and to order them based on the cost from the goal. This eventually leads to sensible node expansions, specially in this example, UC exands much lower number of nodes compared to the previous example, which explains the big difference in RAM and CPU utilisations when comparing both examples.

- Greedy 1

  GR1 is using the first heuristic function, which estimates considerably lower than actual cost values. This makes the agent enqueue and expand nodes that are less likely to be the goal, in a very greedy way. This produces a large number of node expansions, which takes up a big chunk of memory. However, there is an odd behaviour here. GR1 is taking more RAM compared to ID, this might be due to the underestimating heuristic, producing very similar heuristics (due to the healths being almost equal) which forces the agent to expand unnecessary nodes that do not lead to a goal.

- Greedy 2

  GR2 is using the second heuristic function, which also estimates lower than actual cost values, however these estimates are a little closer to the actual cost compared to GR1 heuristic. In this example, using the second heuristic is much more efficient compared to the first one, which forces less node expansions which take less RAM, however the heuristic is more complicated to calculate therefore utilising more RAM than GR1.

- A* 1

  A* 1 is very similar to GR1, however the nodes are expanded using a better estimation compared to using a greedy, heuristic based approach. This causes a fewer number of nodes to be expanded, which in turn uses less memory compared to GR1, however the memory used is still greater than A* 2 because the first heuristic estimates lower values than the second heuristic. However, this also comes at the cost of processing more, by calculating more values (the addition of g(n) and h(n) every time).

- A* 2

A* 2 uses less memory compared to A* 1 because of using the second heuristic function which estimates closer to actual costs, which eventually leads to a lower number of ineffective expansions (expansions that don't reach the goal). It also uses better CPU compared to A* 1 because of the lower number of expansions as well as the lower number of calculations.

4. Number of expanded nodes:

- Breadth First

  The number of expanded nodes in the second grid (B) using BF strategy was 32124, this strategy expanded higher nodes than the previous strategy which is logical since the solution far down the tree. Since this strategy expands on average 4 children per node (with repeated states handled) and as we go deeper in the tree the number of expanded nodes gets a lot higher (4^n).

- Depth First

  The number of expanded nodes in the first grid using DF strategy was 1246, this number is a bit higher than the first grid but still in the same range since the second grid is bigger. As mentioned in the first example the repeated states are handled. Therefore, handling repeated states leads Ethan to the solution faster.

- Iterative Deepening

  The number of expanded nodes in the second grid using ID strategy was 84556, this number was higher than breadth first as expected, this is because it does depth first at different depth limit so it will iterate the same nodes multiple of times until it reaches a depth limit greater than or equal to a solution depth.

- Uniform Cost

  The number of expanded nodes in the second grid using UC strategy was 1201, which is the strategy with the least number of expanded nodes of the uninformed strategies because this strategy always prefers the nodes that cost less.

- Greedy 1

The number of expanded nodes in the second grid using GR1 strategy was 878. This number is much lower than the number of expanded nodes using the uniform cost strategy which is sensible since this strategy is informed which gives a huge advantage.

- Greedy 2

  The number of expanded nodes in the second grid using GR1 strategy was 306. This low number of expanded nodes means that the heuristic function used is very suitable for this grid as it returned a much lower number of expanded nodes than the same strategy using the first function..

- A* 1

  The number of expanded nodes in the second grid using A* 1 strategy was 1234, which is a reasonable result because this strategy depends on an average between the uniform cost and the heuristic cost of Greedy 1.

- A* 2

  The number of expanded nodes in the first grid using A8-2 strategy was 886, as we expected it is less than greedy function 2 and higher than uniform cost, as we explained for A* 1. We also expected that it will be less than A* 1 because the heuristic used in A* 2 is better (as we mentioned before) than the heuristic used in A* 1.