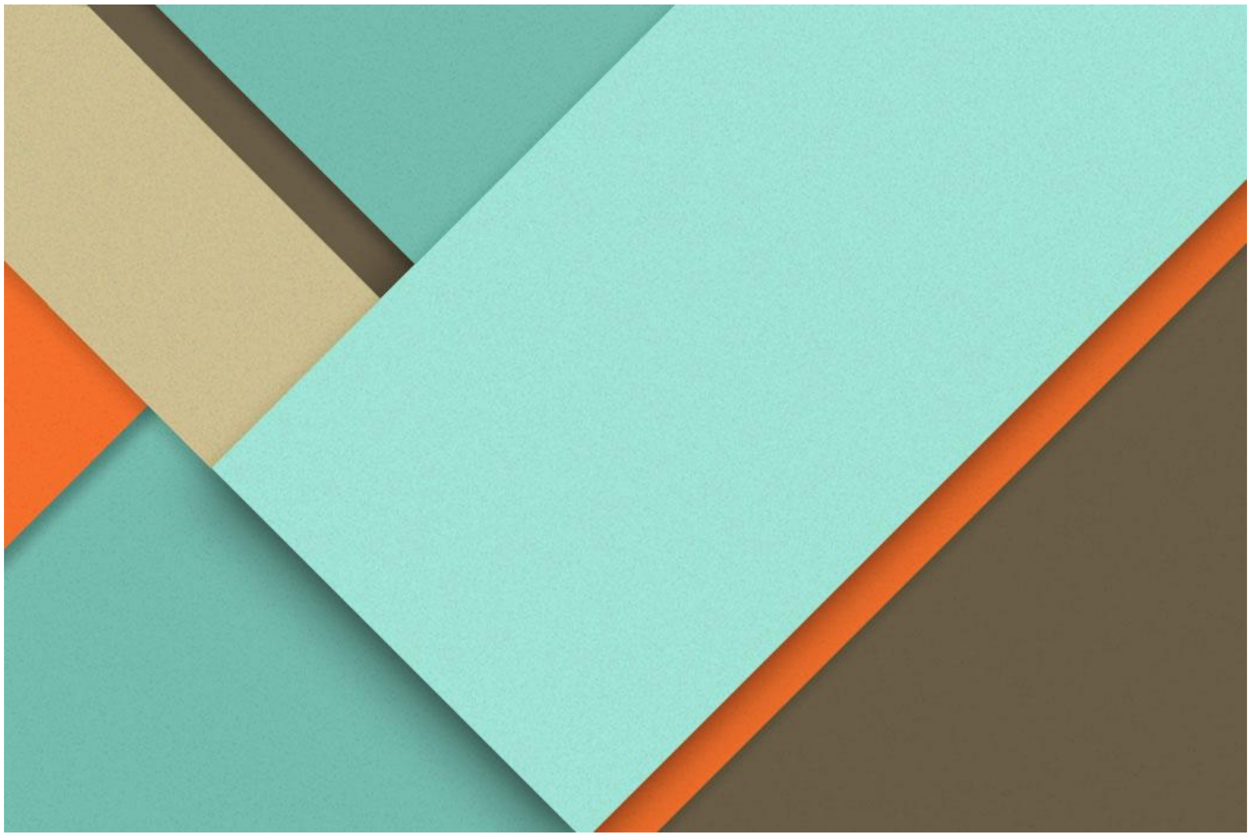

The Matrix: Escaped

CSEN-901 Project Report

Team 3

3.12.2021

—

Eslam Sabry Abdelrehim Mahmoud	43-15786	T17
Bassel Amgad Sharaf	43-6927	T15
Mahmoud Reda Elsayed Elsayed	43-18189	T16
Hussein El Feky	43-1106	T14

Search Tree Node Implementation

We created a public abstract class Node with 5 variables, state, parent,operator, depth, pathCost. The state is a String variable representing the current state of the game with all the needed information. Parent is a Node object representing the parent node to be able to backtrack the sequence of actions that led to this goal. Operator is the action that generated this node from its parent. Depth represents the depth of the node. Finally the pathCost is a variable that will be decided by the path cost function implemented in the Matrix class.

```
public abstract class Node {

    String state;
    Node parent;
    Operator operator;
    int depth;
    int pathCost;

    public Node(String state, Node parent, Operator operator, int depth,
int pathCost) {
        this.state = state;
        this.parent = parent;
        this.operator = operator;
        this.depth = depth;
        this.pathCost = pathCost;
    }
}
```

```
public class MNode extends Node{
    public MNode(String state, MNode parent, Operator operator, int depth,
int pathCost) {
        super(state, parent, operator, depth, pathCost);
    }
}
```

Matrix Problem Implementation

We implemented a **genericSearchProblem** class that has the attributes of the search problem including**:**

- Array of operators that includes all of our operators that can be performed on the node.
- InitialState string.
- hashset visitedStates which includes our expanded state so that we can check for duplicate states and prevent them.
- expandedNodesCnt represents the number of nodes expanded so far.

The class also includes a couple of abstract methods:

1. **goalTestFunction,**
2. **pathCostFunction,**
3. **problemOutput,**
4. **heuristic_1,**
5. **heuristic_2.**

The logic of these abstract methods is implemented by the Matrix class (can also be inherited and implemented by any other search problem).

Furthermore, the genericSearchProcedure method creates the priority queue to store nodes and initializes the hashset object, expandedNodesCnt integer. Afterwards, it starts enqueuing and dequeuing based on the search strategy used.

The class also includes the DepthLimitedSearch method and constructPlan which takes the goal node reached and back tracks to construct a plan.

The Expand method expands the node and adds the children to the queue.

We created the Matrix class that extends the SearchProblem class and inherits the abstract methods and implements the logic. In addition to the abstract methods the Matrix class has the static methods genGrid(), solve(), visualize().

Main Functions

These are the main functions which our code revolves around:

I. genGrid()

The genGrid() method in the matrix class generates a random search problem with random numbers for all variables with respect to their limits and the position of other variables so that no two variables will overlap in the same cell.

II. Solve

The solve method in Matrix class takes in the string grid ,search strategy and visualize boolean as input parameters. It then creates a new Matrix search problem instance with the Matrix constructor and uses the parse method from search strategy class that uses an enum to decide which search method will be used depending on the search strategy given. Then in the matrix class it calls that search strategy method and returns the output.

III. Search

The search method in the search strategy class takes the search problem as an input parameter and then creates a comparator Node depending on the type of the search strategy. For example in the breadth first search we give the Comparator.comparingInt method the node depth as a comparing parameter. The other search methods will be further explained in the next search strategies section. After creating the comparator node it proceeds to call the genericSearchProcedure.

IV. GenericSearchProcedure

The method takes the comparator Node we created in the search method as an input. It proceeds to initialize the visited states hashset to prevent duplicate states, the priorityQueue using the comparator node input and the root node with the initial state. Furthermore, it adds the root to the queue and does a while loop with the condition that the queue is not empty and it can exit if there is a solution found. Inside the loop it dequeues the node from the queue, increases the node count, checks if the state of the dequeued node is goal state if yes the node is given to the ProblemOutput method to return the plan to the given node. else, it expands the node using the expand method in the search problem class.

V. Expand

The method takes the queue, node and visited states hashset as input parameters. It creates a list of operators which is a list of the actions that were previously created inside the Matrix constructor to be used now. The Action class that implements Operator contains the actions as Enums with and each one with its own method.

After creating the operators list it loops on each operator and applies that operator on the state of the current node if the operator returns a state it checks if the state is already present in the hashset if not a new node is created and added to the queue.

VI. ProblemOutput and ConstructPlan

The method in the matrix class takes the goal node as an input parameter, it then creates a state using the state of the node, it then creates the plan string using the constructPlan method in the SearchProblem class. The constructPlan method takes the goal node and loops to its parent until there is no longer a parent. It keeps pushing into the created linked lists the operators,kills,deaths,grid and returns in the end in the reversed order the order the final output which is Final path,deaths,kills,expanded nodes.

Search Algorithms Implementation

As discussed in the methods sections we have an enum class called SearchStrategy that implements an interface called SearchProcedure (SearchProcedure has a method search() which returns a string representing the solution found by the search).

Breadth First Search

1. For the BFS We create a comparator for the Node class using the depth of the node as the comparing unit so that the less the depth the higher priority it has, which achieves the BFS queuing function as the algorithm processes the nodes with smaller depth (the levels higher in the trees) before proceeding to the next level(s).
2. We call the genericSearchProcedure and pass to it the comparator, the comparator represents the queuing function.

Depth First Search

1. For DFS it's the exact opposite of BFS so we give the comparator the negative of the depth so that the higher the depth of the node the more priority it has. Thus, we always take a node having the highest depth in the tree and expand it, enqueue its children (which have higher depth) to the queue. Therefore, the next node we pick is one of the children we just added and so on. This aligns with how DFS works.
2. We call the genericSearchProcedure and pass to it the comparator, the comparator represents the queuing function of the DFS.

Iterative Deepening Search

For the Iterative Deepening Search, we don't use the genericSearchProcedure but we use DepthLimitedSearch instead as follows:

1. In ID we initiate an integer variable called limit. Inside a while loop that keeps calling the DepthLimitedSearch method with the limit
2. The DepthLimitedSearch method operates as follows:
    a. It operates normally like the genericSearchProcedure, having the comparator based on the negative of the depth of the node (similar to DFS).
    b. It keeps track of the maximum depth of a node reached during node expansion.
    c. Whenever we remove a node from the queue, we check that its depth is less than or equal to the limit before checking the goal test or expanding its children. If its depth exceeds the limit, we discard it and process the next

nodes in the queue (in case we discarded it, its depth is not counted towards the maximum depth variable.).

   d. After the queue becomes empty, we check the reason of the queue becoming empty:
      i. If the maximum depth of an expanded node is equal to the limit, then the reason for the queue becoming empty is because we reached the limit and couldn't expand more nodes. In this case, DepthLimitedSearch returns the string "cutoff".
      ii. Otherwise, the reason is there are no more nodes to expand, which means there is no solution found and we couldn't expand the tree more as there are no possible expansions for the leaves. In this case, DepthLimitedSearch returns the string "empty".

3. If the depthLimitedSearch returned "empty" then there is no solution and thus ID returns "No Solution". In case it returns "cutoff" then we increment the limit and the loop performs DepthLimitedSearch again with a higher limit and for the final case a solution is returned.

Uniform Cost Search

In UC we give the comparator of the node the path cost of the node as a comparing parameter. So the lower the cost of the node the higher priority it will have, leading to optimal solutions with lowest possible path costs (minimum deaths possible, ties broken with minimum kills).

Greedy Search

For GR1 and GR2 we use the values given by heuristic_1() and heuristic_2() as a comparing parameter for our node comparator.

A* Search

For AS1 and AS2 we use (heuristic_1 + past cost) and (heuristic_2 + path cost) respectively as a comparing parameter for our node comparator.

Heuristic Functions Used

In order to discuss and explain the heuristics introduced for the Matrix search problem, the used path cost function needs to be elaborated.

Shortest Distance

As a preprocessing step to help carry out the path cost function and the heuristics calculations, floyd-warshall algorithm was run at the beginning of searching to calculate the shortest distance between any 2 cells in the grid (whether the direct path using up, down, right, left faster or an alternative path using pad(s)).

The method `calculate_shortest_distance()` is called inside the constructor of Matrix.

Path Cost Function

- The optimality of any solution is defined, for the Matrix problem, by the number of deaths that solution leads to and any ties in the number of deaths are resolved by the number of kills.
- Therefore, the path cost function should take into account the number of deaths and kills encountered alongside this path.
- Additionally, the path cost function can take into account the individual actions.
- However, the path cost function must satisfy the following constraints:
    - For 2 paths a and b,
        - if deaths (a) < deaths (b),
        - then path_cost_function(a) < path_cost_function(b), regardless of number of kills and number and type of actions of both a and b.
    - For 2 paths a and b,
        - if deaths(a) = deaths(b) and kills(a) < kills(b) ,
        - then path_cost_function(a) < path_cost_function(b), regardless of number and type of actions of both a and b.
    - For 2 paths a and b,
        - if deaths(a) = deaths(b) and kills(a) = kills(b),
        - Then a and b have the same degree of optimality, and thus path_cost(a) and path_cost(b) don't have any constraints.
            - What is meant by the phrase "don't have any constraints" is that in this case whether path_cost(a) are equal or greater or smaller than path_cost(b) won't matter as it wasn't specified how to break ties if deaths(a)=deaths(b) and kills(a)=kills(b) so it can be broken arbitrarily.

- In order to satisfy the previous constraints:
    - Number of kills is multiplied by a weight value that exceeds the maximum possible sum of costs of actions.
    - Number of deaths is multiplied by a weight value that exceeds the maximum possible sum of costs of kills and costs of actions.
    - That guarantees that saving 1 hostage from death is prioritized over saving any number of kills and that refraining from performing a kill action is prioritized over taking any sequence of actions.
- Costs of actions are calculated the sum of
    - Constant value that differs based on the type of action. For example, taking a pill is a favorable action so it has a small constant cost (1), while move actions such as up/left/fly have higher constant cost (4) as they can waste time and lead to hostages dying. The drop action has a relatively small cost (2) as the drop action is favorable action that leads to fully rescuing the hostages.
    - The cost of kills encountered at the time step of performing the action.
        - Doesn't happen except for the case of 'Kill' action.
    - The cost of deaths encountered at the time step of performing the action.
        - This doesn't happen with the 'takePill' action, as the damage of alive hostages doesn't increase during that action.
        - This can happen with all the other actions as with every time step, the damage of all hostages increases by 2.
- The following table summarizes and lists the costs of the actions:

Action	Cost
Up	4 + death_weight * number of deaths encountered along this timestep
Right	4 + death_weight * number of deaths encountered along this timestep
Left	4 + death_weight * number of deaths encountered along this timestep
Down	4 + death_weight * number of deaths encountered along this timestep
Fly	4 + death_weight * number of deaths encountered along this timestep
Carry	4 + death_weight * number of deaths encountered along this timestep
Drop	2 + death_weight * number of deaths encountered along this timestep
TakePill	1 (We don't add cost of deaths because at the time step of taking a pill, the damage of all the alive hostages decreases by 20, so it is impossible to encounter any deaths during this time step. Anyway, we could add that quantity and it won't have any effect.)

Kill	0 + kill_weight * (number of killed agents + number of killed mutants) + death_weight * number of deaths encountered along this timestep

- Death weight and kill weight are fixed throughout any problem. They are calculated once at the beginning of solving any problem and don't change afterwards.
- Kill weight is calculated as the (maximum possible number of actions +1) * maximum cost of an action.
    - The number of actions has an upper bound because in our implementation, the search tree doesn’t expand nodes with repeated states, so eventually and because of the damage of hostages increasing with every time step and having a fixed number of pills, the tree will always hit a deadwall and be not able to expand more nodes as any possible expansion will result in a node with either an invalid state or a repeated state that has been expanded before elsewhere in the tree.
    - This upper bound is calculated by the method `est_MaxDepth_sol()` at the beginning of solving the problem and doesn’t change afterwards.
        - The method `est_MaxDepth_sol()` calculates an upper bound of the number of performed actions in any solution or the maximum depth of a solution for some Matrix search problem.
        - The deepest solution is one that has all the hostages dying after taking as many pills as possible (all the pills available) and then Neo kills them all and returns to the telephone booth. The reason that this solution is deeper than any solution that saves at least one hostage is because saving a hostage (carrying the hostage, going to the booth and then dropping them) requires Neo to perform and finish all these steps before that hostage(s) damage reaches 100; which means that it had performed these steps in a smaller number of steps, i.e: that solution has a shallower depth.
        - After that (all the hostages dying after taking all possible pills), Neo will have to go to each hostage to kill him/her. We can overestimate the number of steps required for that as the maximum of [shortest distance from Neo’s initial location to the hostage’s location + 1 + shortest distance from the hostage’s location to the telephone booth location + 1] over all the hostages. The two “+1”s are to account for any longer distance as we go to a neighbouring cell to the mutant to kill it and not to its exact cell.
        - The repeated states prevents the agent from looping aimlessly increasing the maximum depth.

    - With the configuration listed in the table above, the maximum cost of an action = 4. (We don't consider the cost coming from the kills or deaths encountered in the action costs here.)
- Death weight is calculated as the (maximum possible number of kills + 1) * kill_weight
    - Maximum possible number of kills for any Matrix problem is the initial number of hostages + the initial number of agents.
    - We multiply by the kill_weight as well to make sure that 1 death is costlier than any number of actions.

Heuristic_1

The first admissible heuristic used is calculated as the sum of the following factors of the state of the node we are calculating the heuristic for:

1. Number of mutated hostages (not yet killed) * kill_weight
2. Shortest distance between Neo's current position and the telephone booth
3. If neo is carrying 1 or more hostages, we add the cost of the drop action. (We add it once regardless of the number of carried hostages as we perform the drop only once for any n>=1 carried hostages.)
4. Number of hostages (still alive and not carried) * cost of Carry action

This heuristic is admissible because:

1. At a goal state all the mutated hostages are already killed. Thus, any mutated hostage that has not been killed yet will have to be killed to reach a goal state. Thus, We are not overestimating as this term will always be added to the path cost in order to reach a goal state.
2. All goal states must have Neo at the telephone booth. Thus, the cost of actions that lead to Neo's reaching the goal state must be added to the path cost to reach a goal state. We are not overestimating because we use the "shortest distance" here which Neo can travel that distance exactly or take a longer path. Additionally, Neo may need to perform some more actions which will add to the total path cost. Thus, we are not overestimating.
3. Any carried hostage will have to be dropped at the telephone booth in order to reach a goal state. Thus, this term will always be added to the final path cost. Therefore, we are not overestimating.
4. For any hostage still alive and not carried yet, there is only one of two possible outcomes:

a. The hostage dies before carrying it, which means that it becomes mutated and therefore Neo will have to kill it. The cost for that kill action is much greater than the cost of ‘Carry’ action (cost of kill_action = no_of_killed * kill_weight where kill_weight=(max_depth_of_tree * max_cost_of_operator where max_depth_of_tree is greater than or equal to one for any non-trivial problem). So, by using only the cost of ‘carry’ action, we are not overestimating.
b. The hostage doesn’t die and Neo carries it before dying. The cost for this outcome will probably be the cost of movements to reach the hostage + cost of carry + cost of moving to the telephone booth. We are only counting the cost of ‘carry’ in our heuristic so we are not overestimating.

Therefore, as each term we add up in our heuristics is not overestimating, heuristic_1 is not overestimating.

Heuristic_2

The admissible heuristic used is calculated as the sum of the following factors of the state of the node we are calculating the heuristic for:

1. Number of mutated hostages (not yet killed) * kill_weight
2. We have one of two cases:
    a. If there are hostage still alive and were not carried by Neo yet:
        i. iterate on the hostages and calculate variable X for each hostage where
        ii. X = shortest distance from Neo’s location to the hostage’s location + shortest distance from that hostage’s location to the telephone booth
        iii. Take the hostage with the maximum X

        We add this X to our heuristics.
    b. Otherwise, we add the shortest distance from Neo to the telephone booth to our heuristics.
3. If Neo is carrying 1 or more hostages, we add the cost of the drop action. (We add it once regardless of the number of carried hostages as we perform the drop only once for any n>=1 carried hostages.)

This heuristic is admissible because:

1. At a goal state all the mutated hostages are already killed. Thus, any mutated hostage that has not been killed yet will have to be killed to reach a goal state. Thus, We are not overestimating as this term will always be added to the path cost in order to reach a goal state.
2. In either cases a or b, we calculate the distance between Neo and the telephone booth only once, which will always be added in the path cost function.

        a. For all of these alive hostages that were not carried,
            i. We will either save them by carrying them and dropping them at the telephone booth. For this case, we calculated only the cost of movements for saving the furthest hostage out of them, which is an underestimate because
                1. we will have to perform the carry action for each hostage that we save + one drop action for each batch of carried hostages + probably the other hostages (if there are) will require Neo to move additional moves to reach and carry them, or Neo's carry capacity may not be enough to carry all hostages at once and then he will need to travel back and forth between the telephone booth and the hostages. We estimate all of these factors by only the maximum movements required for one hostage only; which is clearly an underestimate.
                2. That hostage will die, and then we will need to go to one neighbouring cell of his and perform the kill action. We counted movements to the hostage's exact location, which can be greater than the actual distance traveled to reach a neighbouring cell by 1, but then we didn't calculate the cost of the kill action which is clearly far greater than 1 (cost of kill_action = no_of_killed * kill_weight where kill_weight=(max_depth_of_tree * max_cost_of_operator where max_depth_of_tree is greater than or equal to one for any non-trivial problem).

3. Any carried hostage will have to be dropped at the telephone booth in order to reach a goal state. Thus, this term will always be added to the final path cost. Therefore, we are not overestimating.

Therefore, as each term we add up in our heuristics is not overestimating, heuristic_2 is not overestimating.

Examples

I. Example Output Grid 3

- **Breadth First**

  Kill,left,kill,left,left,down,kill,down,left,carry,right,down,kill,left,down, takePill,up,right,right,kill,right,right,down,drop;5;10;5812

- **Depth First**

  Kill,down,down,fly,down,left,kill,up,right,fly,up,left,kill,left,left,left, down,carry,up,right,right,right,down,right,fly,down,drop,left,left,kill,lef t,left,takePill,right,up,right,right,up,up,left,kill,up,right,right,down,dow n,fly,down;5;8;2416

- **Uniform Cost**

  kill,down,left,down,down,down,kill,left,kill,left,left,takePill,right,right, right,up,up,up,kill,left,left,left,down,carry,up,right,right,kill,down,down , down,right,right,drop;5;8;8425

- **Iterative Deepening**

  Kill,left,kill,left,left,down,kill,left,down,carry,right,down,kill,left,down, takePill,up,right,right,kill,right,down,right,drop;5;10;30976

- **Greedy 1**

  kill,down,down,fly,down,left,kill,left,kill,left,left,takePill,right,right, right,up,up,up,kill,left,kill,left,left,down,carry,up,right,right,down,down ,down,right,right,drop;5;8;1178

- **Greedy 2**

  Kill,down,left,kill,left,left,left,down,carry,up,right,right,right,down, down,down,kill,right,drop,left,left,kill,left,left,takePill,right,right,right, up,up,up,left,kill,down,down,down,right,right;5;8;4286

- **A* 1**

  kill,down,left,kill,down,right,fly,down,left,kill,left,kill,left,left,takePill,ri ght,right,right,up,up,up,left,kill,left,left,down,carry,up,right,right,down, down,down,right,right,drop;5;8;2641

- **A* 2**

  kill,down,left,kill,down,down,down,kill,left,kill,left,left,takePill,right,up,right,right,up,up,left,kill,left,left,down,carry,up,right,right,down,right,right,fly,down,drop;5;8;2557

II. Example Output Grid 8

- **Breadth First**

  left,left,carry,left,carry,right,up,up,drop,right,up,carry,left,left,up,carry,down,right,down,drop;0;0;52767

- **Depth First**

  up,up,up,kill,up,kill,down,down,down,down,left,left,carry,up,takePill,up,drop,kill,up,kill,up,left,carry,down,down,down,down,carry,up,up,up,up,right,down,kill,up,left,down,down,down,down,right,right,up,takePill,up,left,drop;2;5;69

- **Uniform Cost**

  Left,left,carry,left,carry,right,up,up,drop,up,left,up,carry,down,right,right,carry,down,left,drop;0;0;2901

- **Iterative Deepening**

  Up,up,up,left,carry,left,kill,up,left,carry,down,right,down,drop,down,takePill,down,left,carry,right,carry,up,up,drop;0;1;164516

- **Greedy 1**

  up,up,left,left,up,kill,down,down,takePill,up,up,right,carry,down,left,drop,kill,down,down,carry,up,up,drop,down,right,takePill,up,left,up,up,left,carry,down,down,right,drop,down,left,down,carry,up,up,right,drop;0;2;160

- **Greedy 2**

  Up,up,left,left,up,kill,down,down,left,down,carry,up,kill,up,right,drop,up,up,right,kill,down,carry,left,down,drop,down,takePill,up,up,up,left,carry,down,down,down,down,right,carry,up,up,drop;2;3;263

- **A* 1**

  left,left,carry,left,carry,up,right,up,drop,right,up,carry,left,left,up,carry,down,right,down,drop;0;0;1584

- **A* 2**

  Left,left,carry,left,carry,up,right,up,drop,up,left,up,carry,down,right,right,carry,down,left,drop;0;0;804

III. Visualized Examples

Symbols used in the visualization is summarized by the following table:

Symbol	Meaning
N	Neo's current location
T	Telephone Booth
A	Agent
H(xx)	An alive hostage's location, with their current damage between the brackets
C(xx)	A carried hostage with their current damage between the brackets, this symbol will always appear in the same cell Neo's currently at. When Neo drops the carried hostages, these symbols will disappear as they escape the matrix.
M	A mutated hostage (a hostage that died before being carried/rescued) that has not been killed yet by Neo.
P	Pill (not taken yet)
FxTy	Pad. Because pads are bidirectional, we treat a pair of pads that are bidirectional as if they are 2 separate pads, one goes from A to B and the other from B to A.
	so for 2 pads between (0,0) and (3,1). Pad 0 will be F0 (from) at (0,0) and T0 (to) at (3,1) and Pad 1 will be F1 (from) at (3,1) and T1 (to) at (0,0).

The visualization lines order is as follows:

```
Grid representation at state 1
Number of deaths, kills and Neo's damage at state 1
action from state 1 to state 2
Grid representation at state 2
Number of deaths, kills and Neo's damage at state 2
action from state 2 to state 3
…
…
…
action from state n-1 to state n
Grid representation at state n
Number of deaths, kills and Neo's damage at state n
```

In order to make reading the report easier, we uploaded the running examples visualized output in txt files that can be accessed by the following urls:

Grid 3	AS1	https://drive.google.com/file/d/1-z7Bddc0KLaSZEoqdZK_hgx6gJIYtIOd/view?usp=sharing
Grid 3	UC	https://drive.google.com/file/d/1CHz3B6oZVMRXN0YfvE9qO4nAhsMcrQPN/view?usp=sharing
Grid 5	BF	https://drive.google.com/file/d/1wQKnB_WWRvKlKv05uDn6dcjMRzSDpGj1/view?usp=sharing
Grid 8	AS1	https://drive.google.com/file/d/17WQyqi_dOpqUNHCFxRiKrTqWPH8e65L1/view?usp=sharing

Or you can access the folder containing the examples here:

https://drive.google.com/drive/folders/1zN5Wga2SfpGTUjbmM97TeoEWIdOcouDO?usp=sharing

Comparison of the performance

We used the Java Operating System MXBean library to monitor the CPU and Memory utilization of the running process for each of the grids.

	Grid 3			Grid 8		
	CPU	**MEM**	**Nodes**	**CPU**	**MEM**	**Nodes**
BF	39.17%	1.02%	5812	9.31%	6.00%	52767
DF	7.25%	0.72%	2416	8.90%	0.25%	69
UC	15.06%	1.47%	8425	37.91%	0.99%	2901
ID	8.51%	2.50%	30976	8.17%	5.30%	164516
GR1	17.03%	0.73%	1178	15.06%	0.42%	160
GR2	30.13%	1.43%	4286	7.53%	0.50%	263
AS1	46.38%	1.06%	2641	37.90%	1.39%	1584
AS2	35.60%	1.01%	2557	15.67%	0.78%	804

	Grid 3		Grid 8	
	Deaths	**Kills**	**Deaths**	**Kills**
BF	5	10	0	0
DF	5	8	2	5
UC	5	8	0	0
ID	5	10	0	1
GR1	5	8	0	2
GR2	5	8	2	3
AS1	5	8	0	0
AS2	5	8	0	0

I. Completeness

All of the implemented search algorithms are complete since we created a hashset to prevent repeated states. Therefore, whenever there is a solution all the algorithms will be able to find it. If there doesn't exist a solution, the search algorithm will exhaust all expandable nodes until it cannot expand any more nodes and this will mean that there doesn't exist a solution.

This check of repeated states made the incomplete search strategies (Depth first search and Greedy) complete in this specific implementation and search problem.

II. Optimality

For our examples (grid 3 and grid 8), the optimal search strategies (Uniform Cost and A*) give the same optimal number of deaths and kills which are 5, 8 and 0, 0 for grid3 and grid 8 respectively.

- **BF** is known to be a non-optimal search strategy as it returns the first solution it finds during its First-In-First-Out queuing fashion and doesn't take the cost into consideration. We can see this as
  a. Grid 3: it gives a non-optimal solution where the number of deaths is the same as in the optimal solution but has a bigger number of kills which means it is not an optimal solution.
  b. Grid 8: it gives an optimal solution for grid 8 which has the same number of deaths and kills as the optimal solutions. This happened because an optimal solution was the first solution to be encountered.
- **DF** is known to be a non-optimal search strategy as it returns the first solution it finds during its Last-In-First-Out queuing fashion and doesn't take the cost into consideration. We can see this as
  a. Grid 3: It gives an optimal solution which has the same number of deaths and kills as the optimal solutions. This happened because an optimal solution was the first solution to be encountered.
  b. Grid 8: it gives a non-optimal solution where the number of deaths is greater than the number of deaths in the optimal solution which means it is not an optimal solution.
- **UC** is known to be an optimal search strategy as it always picks the node that has the minimum path cost. We can see this as:
  a. Grid 3: It gave the optimal answer which is 5 deaths and 8 kills. There doesn't exist a solution for grid 3 that gives less kills or less deaths.
  b. Grid 8: It gave the optimal answer which is 0 deaths and 0 kills, which is the absolute optimal for a Matrix search problem.
- **ID** is known to be a non-optimal search strategy as it returns the first solution it finds during its expansion. We can see this as:
  a. Grid 3: it gives a non-optimal solution having 5 deaths but 10 kills which is greater than the 5 deaths, 8 kills optimal solution.
  b. Grid 8: it gives a non-optimal solution having 0 deaths but 1 kills which is greater than the 0 deaths, 0 kills optimal solution.
- **Greedy strategy** is known to be a non-optimal search strategy as it expands nodes based on just heuristics which is just an estimate of the cost but doesn't take actual cost into consideration. We can see this as:
  a. Greedy using heuristics_1
     i. Grid 3: it gave an optimal solution (5 deaths 8 kills) which means that heuristc_1 was able to guide the search algorithm to find one optimal solution, which is not always the case.
     ii. Grid 8: it gave a non-optimal solution (0 deaths but 2 kills) which means that heuristic_1 failed to guide the search algorithm to find an optimal solution.

b. Greedy using heuristics_2
    i. Grid 3: it gave an optimal solution (5 deaths 8 kills) which means that heuristc_1 was able to guide the search algorithm to find one optimal solution, which is not always the case.
    ii. Grid 8: it gave a non-optimal solution (2 deaths 3 kills) which means that heuristic_1 failed to guide the search algorithm to find an optimal solution.

- A* strategy is known to be an optimal search strategy as it always picks the node that has the minimum (path cost+heuristics). We can see this as:
    a. A* using heuristics_1
        i. Grid 3: it gave an optimal solution (5 deaths 8 kills).
        ii. Grid 8: it gave an optimal solution (0 deaths 0 kills).
    b. A* using heuristics_2
        i. Grid 3: it gave an optimal solution (5 deaths 8 kills).
        ii. Grid 8: it gave an optimal solution (0 deaths 0 kills).

III. RAM Usage

For Grid3 most algorithms were close to each other in terms of memory usage, which can happen depending on the grid and how the goal node is similarly reached for different algorithms. However, the algorithm that used the most memory was Iterative deepening and if we look at its number of nodes we will find that it's much bigger than the rest of the algorithms.

For Grid8 we can see from the table that the search algorithms that used the most memory are Breadth First and Iterative deepening which is to be expected since they keep expanding all the nodes level by level until finding the goal state. Furthermore, the least memory was used by Depth First because it has an O(b*d) space complexity which is much less than the rest of the algorithms, not to mention that we removed repeated states which helped the depth first be complete and perform better.

IV. CPU Utilization

For Grid 3, ID and DF had the lowest CPU utilization at around 8%. Following them are UC and GR1 at about 16%. Then come GR2 at 30%, AS2 at 36%, BF at 39% and finally AS1 at 46.38%.

For Grid 8, ID, DF, GR2 and BF had the lowest CPU utilization at around 8.4%. Following them are AS2 and GR1 at about 15.4%. Then come AS1 and UC at 37.9%

From these results, we can see that:

- ID and DF had the lowest CPU utilization with both grids.
- AS1 had the biggest CPU utilization with both grids.
- Other strategies had varying amounts in the middle.
    - While BF had one of the biggest CPU utilizations with grid 3, it had one of the lowest CPU utilizations with grid 8.

V. Number of expanded nodes

Grid 3:

- We can see that in terms of number of expanded nodes:
    - GR1 < DF < AS2 < AS1 < GR2 < BF < UC < ID

Grid 8:

- We can see that in terms of number of expanded nodes:
    - DF < GR1 < GR2 < AS2 < AS1 < UC < BF < ID

These results show that:

- Iterative Deepening had expanded the greatest number of nodes in both examples. This can be justified by the fact that ID tends to visit the nodes of all the levels but the last level that contains the returned solution.
- Uniform Cost and Breadth-First were just after the Iterative Deepening in terms of greater number of expanded nodes.
- Depth First Search and Greedy using heuristics_1 had expanded the least number of nodes in both examples.
- A* using either heuristics expanded a smaller number of nodes than the Uniform Cost Search. This aligns with the fact that a* strategy using an admissible heuristics (like heuristic_1 and heuristic_2) is optimally efficient (meaning that in comparison with any optimal search strategy such as Uniform Cost, a* will expanda number of nodes less than or equal to that of the other optimal strategy).

- Heuristic_2 seems to lead the A* strategy to expand less nodes than heuristic_1, while it leads the Greedy strategy to expand more nodes than heuristic_1.
- Uniform Cost search has one of the biggest numbers of expanded nodes among the implemented strategies. However, this pays off as UC gives optimal solutions unlike other non-optimal strategies that expand less nodes such as greedy.

References

https://stackoverflow.com/questions/19781087/using-operatingsystemmxbean-to-get-cpu-usage

https://stackoverflow.com/questions/17374743/how-can-i-get-the-memory-that-my-java-program-uses-via-javas-runtime-api

https://docs.oracle.com/javase/7/docs/jre/api/management/extension/com/sun/management/OperatingSystemMXBean.html#getFreePhysicalMemorySize()