Project 1 Report

- We have in our implementation an abstract Node class representing each node in the search tree. It has the following properties: a string called 'state' representing the state of that node, a Node object called 'parentNode' representing the parent node to that node (the root's parentNode is null), a string called 'operator' representing the operator that got us to that node, an integer called 'depth' representing the depth of that node, and finally an int called 'pathCost' representing the path cost of that node. We have a MatrixNode class that extends Node. This is the class we use in Matrix.
- We have a Matrix class containing the main method and all methods we used to solve the search problem. Matrix extends an abstract class called SearchProblem. Matrix has the following properties which are the properties of a SearchProblem: a list called 'operators' containing the operators of our problem (fly, right, carry, etc.), initialState which is a string representing the initial state of our grid, a hash map called 'generatedStates' containing all the states generated as keys to check for repeated states, an int called 'chosenNodes' to represent the number of nodes chosen for expansion. Matrix also has two methods inherited from SearchProblem, goalTest(String state) to check if the current state is a goal state, and pathCost(String state) to calculate the path cost of a certain node using its state.
- In our implementation, we created several methods to be able to solve the search problem. These methods are:-
    1- boolean goalTest(String state): To check if a state is a goal state (node) or not. The conditions we check for are that Neo is in the same location as the telephone booth, no hostages are left, either carried or on their own, and no mutated agents are alive.

2- int pathCost(String state): To calculate the path cost of a node using its state using the following rule, (225*deaths) + kills.
3- String genGrid(): To generate the grid as described in the project description.
4- String generateInitialState(String grid): To generate the initial state using the initial grid.
   Our state takes the following form:
   neoX,neoY,neoDamage,C;M,N;hostage1X,hostage1Y, Hostage1Damage,.....;carriedHostage1Damage,.....;pill1X, pill2X,.....;agent1X,agent1Y,.....;mutatedAgent1X, mutatedAgent1Y,.....;pad1StartX,pad1StartY,pad1FinishX, pad1FinishY,.....;telephoneX,telephoneY;deaths;kills
   where deaths and kills are integers represents the number of hostages dead and agents killed repectively.
5- Node makeNode(String state, Node parentNode, String operator, int depth, int pathCost): To make a Node object from a previous state after applying an operator to it.
6- boolean isApplicable(String state, String operator): To check if applying an operator to a state is possible or not.
7- String transform(String state): To transform a state to another form of this state in order to check for repeated states. This transformation removes the damage of hostages, C, deaths and kills, and replaces carriedHostages damage by the number of carried hostages.

8- boolean isRepeated(String newState): To check if a state is repeated or not, in order to avoid repeated states.

9- String act(String state, String operator): To generate the new state resulting from applying ‘operator’ to ‘state’, given that operator is applicable to the state.

10- int heuristicOne(String state): To calculate the value of our first heuristic function when applied to a node using its state. The heuristic is described later in the report.

11- int heuristicTwo(String state): To calculate the value of our first heuristic function when applied to a node using its state. The heuristic is described later in the report.

12- ArrayList<Node> expand(Node node, List<String> operators, String strategy): To expand a node by applying the applicable operators to it and store the resulting states in the queue according to the strategy used.

13- String visualize(String state, String operator): To visualize the grid and the effect of Neo’s actions.

14- String[] goalBackTrack(Node node): To back track the goal node in order to construct the plan.

15- String depthLimitedSearch(Node root, int depth, boolean visualize): To construct the solution of iterative deepening search at each depth.

16- String solve(String grid, String strategy, boolean visualize): It is the most important method that uses all previous methods to solve the search problem.

- We used a priority queue to implement different search algorithms. In the case of BFS, each node has the path cost of its parent + 1, while it has the path cost of its parent – 1 in the case of DFS. In the case of uniform cost search, we use a path cost function as the path cost, which is (225*deaths)+kills. In greedy search we use the heuristics implemented, and in the A* search we use the heuristics added to the normal path cost function. Iterative deepening is implemented in a similar way to DFS but starting with a maximum depth of zero and incrementing it by 1 each step until we reach the goal.
- The first heuristic we implemented estimates the number of deaths (225*numOfDeaths). We assume that the distance between Neo and uncarried hostages to be zero, and we assume that the telephone booth is at a neighboring cell to Neo, so, the number of steps we need to execute to save the hostage will be 2 (carry and move to the booth), and in the case of carried hostages we assume that Neo is in a neighboring cell to the telephone booth, so the number of steps to save the carried hostages will be 1 (move to the booth), if one hostage's damage will reach 100 at these number of steps, we assume this hostage dead, this heuristic is admissible as we assume that Neo is in the same cell as the hostage and Neo is next to the telephone booth which might not be the case in the actual state. The second heuristic estimates both the number of deaths and kills (225*numOfDeaths) + kills. Death estimation works as in the first heuristic. To estimate the number of kills we assume that we only need to kill mutated agents and that we will not kill normal agents, this heuristic is admissible as we assume that we only kill mutated agents and that is the best case in the actual simulation, as we normally may need to kill normal agents in addition to all mutated agents.

- First grid: 5,5;2;0,4;1,4;0,1,1,1,2,1,3,1,3,3,3,4;1,0,2,4;0,3,4,3,4,3,0,3;0,0,30,3,0,80,4,4,80

  BF:
  left,left,kill,left,left,carry,down,down,down,down,kill,right,right,right,kill,fly,right,down,drop;2;4;56286

      Complete
      Not optimal
      Memory Utilization: 6.700387084439773 %
      CPU Utilization: 30 %
      Expanded Nodes: 56286

  DF:
  down,left,down,left,down,down,left,left,up,carry,up,up,takePill,up,carry,down,down,down,down,right,right,up,up,up,up,right,right,down,drop,up,left,fly,left,up,up,up,kill,up,right,fly,left,up,kill,up,up,up,right,fly,kill,fly,down,down,left,down,left,left,up,up,up,kill,down,down,down,down,right,right,up,up,up,up,right,right,down;2;5;77

      Complete
      Not optimal
      Memory Utilization: 0.24819898795792592 %
      CPU Utilization: 10 %
      Expanded Nodes: 77

UC:
left,fly,left,left,left,up,carry,up,up,takePill,down,down,down,right,right,up,up,right,right,takePill,up,up,left,fly,right,carry,left,fly,down,right,drop,up,left,fly,left,left,left,up,up,up,up,carry,down,down,down,down,right,right,right,fly,down,right,drop;0;0;432

Complete

Optimal

Memory Utilization: 0.7404116719998106 %

CPU Utilization: 10 %

Expanded Nodes: 432

ID:
down,down,kill,up,up,left,left,kill,left,left,carry,down,down,kill,down,down,right,right,right,kill,fly,down,right,drop;2;6;241642

Complete

Not optimal

Memory Utilization: 13.697769969902884 %

CPU Utilization: 45 %

Expanded Nodes: 241642

GR1:
```
down,left,down,kill,right,kill,takePill,down,left,left,down,left,left,u
p,up,up,takePill,down,down,carry,down,right,right,right,right,carr
y,up,up,up,drop,left,left,kill,left,kill,left,down,down,kill,right,right,
right,down,fly,left,left,left,carry,right,right,right,right,down,drop;0
;6;3131
```
Complete
Not optimal
Memory Utilization: 0.2396857142530514 %
CPU Utilization: 13 %
Expanded Nodes: 3131

GR2:
```
down,left,down,kill,right,kill,takePill,down,left,left,down,left,left,u
p,up,up,takePill,down,down,carry,down,right,right,right,right,carr
y,up,up,up,drop,left,left,kill,left,kill,left,down,down,kill,right,right,
right,down,fly,left,left,left,carry,right,right,right,right,down,drop;0
;6;3131
```
Complete
Not optimal
Memory Utilization: 0.2703646854368587 %
CPU Utilization: 14 %
Expanded Nodes: 3131

AS1:

down,down,takePill,up,up,left,fly,left,left,left,up,up,up,takePill,down,down,carry,down,right,right,right,right,carry,left,fly,down,right,drop,left,up,fly,left,left,left,up,up,up,up,carry,down,down,down,down,right,right,right,fly,down,right,drop;0;0;316

Complete

Optimal

Memory Utilization: 1.6570704483391578 %

CPU Utilization: 12 %

Expanded Nodes: 316

AS2:

down,down,takePill,up,up,left,fly,left,left,left,up,up,up,takePill,down,down,carry,down,right,right,right,right,carry,left,fly,down,right,drop,left,up,fly,left,left,left,up,up,up,up,carry,down,down,down,down,right,right,right,fly,down,right,drop;0;0;316

Complete

Optimal

Memory Utilization: 1.692327896830563 %

CPU Utilization: 12 %

Expanded Nodes: 316

- For the first grid, the order of search strategies ascendingly according to the CPU utilization came as follows:
    1- DF, UC: 10 %
    2- AS1, AS2: 12 %
    3- GR1: 13 %
    4- GR2: 14 %
    5- BF: 30 %
    6- ID: 45 %

  So, DF and UC are the strategies that have the least CPU usage, while ID is the most strategy that uses CPU.

- For the first grid, the order of search strategies ascendingly according to the RAM utilization came as follows:
    1- GR1: 0.2396857142530514 %
    2- DF: 0.24819898795792592 %
    3- GR2: 0.2703646854368587 %
    4- UC: 0.7404116719998106 %
    5- AS1: 1.6570704483391578 %
    6- AS2: 1.692327896830563 %
    7- BF: 6.700387084439773 %
    8- ID: 13.697769969902884 %

  So, GR1 is the strategy that has the least RAM usage, while ID is the most strategy that uses RAM.

- For the first grid, the order of search strategies ascendingly according to the number of expanded nodes came as follows:
    1- DF: 77
    2- AS1, AS2: 316
    3- UC: 432
    4- GR1, GR2: 3131
    5- BF: 56286
    6- ID: 241642

  So, DF is the strategy that expands the most nodes, while ID is the most strategy that expands the least number of nodes.

- Second grid: 5,5;3;1,3;4,0;0,1,3,2,4,3,2,4,0,4;3,4,3,0,4,2;1,4,1,2,1,2,1,4,0,3,1,0,1,0,0,3;4,4,45,3,3,12,0,2,88

  BF:
  up,left,carry,down,right,down,down,carry,right,down,kill,carry,left,left,left,left,drop;1;1;53647
    Complete
    Not optimal
    Memory Utilization: 9.386812164638673 %
    CPU Utilization: 32 %
    Expanded Nodes: 53647

DF:

up,fly,up,kill,down,fly,down,down,down,carry,up,up,up,fly,up,right,kill,down,down,down,down,left,drop,up,takePill,up,up,fly,down,down,down,right,takePill,down,carry,up,left,up,up,up,fly,down,down,down,drop;1;2;49

Complete
Not optimal
Memory Utilization: 0.21161992365032364 %
CPU Utilization: 15 %
Expanded Nodes: 49

UC:

up,fly,down,down,takePill,right,down,right,takePill,left,up,up,right,right,down,right,takePill,down,carry,up,left,carry,up,up,up,left,carry,down,down,left,down,down,left,drop;0;0;557

Complete
Optimal
Memory Utilization: 0.881455755975847 %
CPU Utilization: 14 %
Expanded Nodes: 557

ID:
right,kill,down,down,down,carry,up,left,carry,up,up,up,kill,fly,down,down,down,drop;1;3;94867

Complete

Not optimal

Memory Utilization: 11.679662198275677 %

CPU Utilization: 40 %

Expanded Nodes: 94867

GR1:
down,down,right,takePill,down,kill,left,left,kill,right,up,carry,down,left,takePill,up,left,left,takePill,right,up,right,right,kill,left,up,up,carry,down,fly,down,down,left,down,left,left,left,drop,right,right,right,right,carry,left,left,left,left,up,up,right,up,kill,down,right,down,down,left,left,drop;0;4;622

Complete

Not optimal

Memory Utilization: 0.9167050087259847 %

CPU Utilization: 14 %

Expanded Nodes: 622

GR2:

```
down,down,right,takePill,down,kill,left,left,kill,right,up,carry,dow
n,left,takePill,up,left,left,takePill,right,up,right,right,kill,left,up,up,
carry,down,fly,down,down,left,down,left,left,left,drop,right,right,
right,right,carry,left,left,left,left,up,up,right,up,kill,down,right,do
wn,down,left,left,drop;0;4;622
```

Complete
Not optimal
Memory Utilization: 0.9167047985787727 %
CPU Utilization: 15 %
Expanded Nodes: 622

AS1:

```
down,down,right,takePill,down,carry,up,left,carry,up,left,left,dow
n,left,takePill,down,drop,right,right,takePill,left,up,up,up,right,up,
carry,right,fly,down,down,down,drop;0;0;323
```

Complete
Optimal
Memory Utilization: 0.6698936228088203 %
CPU Utilization: 15 %
Expanded Nodes: 323

AS2:
down,down,right,takePill,down,carry,up,left,carry,up,left,left,down,left,takePill,down,drop,right,right,takePill,left,up,up,up,right,up,carry,right,fly,down,down,down,drop;0;0;323

Complete

Optimal

Memory Utilization: 0.7051498104169536 %

CPU Utilization: 13 %

Expanded Nodes: 323

- For the second grid, the order of search strategies ascendingly according to the CPU utilization came as follows:

    1- AS2: 13 %
    2- UC, GR1: 14 %
    3- DF, AS1, GR2: 15 %
    4- BF: 32 %
    5- ID: 40 %

  So, AS2 is the strategy that has the least CPU usage, while ID is the most strategy that uses CPU.

- For the second grid, the order of search strategies ascendingly according to the RAM utilization came as follows:
    1- DF: 0.21161992365032364 %
    2- AS1: 0.6698936228088203 %
    3- AS2: 0.7051498104169536 %
    4- UC: 0.881455755975847 %
    5- GR2: 0.9167047985787727 %
    6- GR1: 0.9167050087259847 %
    7- BF: 9.386812164638673 %
    8- ID: 11.679662198275677 %

  So, DF is the strategy that has the least RAM usage, while ID is the most strategy that uses RAM.

- For the second grid, the order of search strategies ascendingly according to the number of expanded nodes came as follows:
    1- DF: 49
    2- AS1, AS2: 323
    3- UC: 557
    4- GR1, GR2: 622
    5- BF: 53647
    6- ID: 94867

  So, DF is the strategy that expands the most nodes, while ID is the most strategy that expands the least number of nodes.

Resources:

- https://stackoverflow.com/questions/37916136/how-to-calculate-memory-usage-of-a-java-program
- https://www.tutorialspoint.com/how-to-check-the-memory-used-by-a-program-in-java