AI Project 1 Report

1) <u>Implementation of the search-tree node</u>

Attributes:

- State: the state the this node corresponds to
- Parent: the parent node of this node
- Action: the action applied to get this node
- Pathcost: the cost from root to this node
- Heuristic value: the heuristic value assigned to this node (if applicable to the search strategy)
- Depth: the depth of this node in the tree

Methods:

- **Constructor1:** initializes an instance of anode using a given set of attributes.
- **Constructor2:** initializes an instance of a node by setting all the attribute values to nulls and zeros.
- **Attributes getters and setters**
- **Compareto:** compares between the current instance of the node and a given node to sort them by checking the path cost of the two nodes in case the search strategy is uniform cost or by checking the heuristic value if the search strategy is greedy or the combination of the path cost and the heuristic value incase the search strategy is A*.

2) <u>Implementation of the Matrix problem</u>

The *matrix* class, which is used to solve the Matrix problem, was created as a subclass of the *searchProblem* class (representing the generic search problem). The *searchProblem* class contains a *matrixState* attribute that stores the problem’s initial state along with four abstract methods (actions, result, isGoal and stepCost). The single attribute along with the 4 abstract methods are inherited and utilized by the *matrix* class. Below is a somewhat detailed description of the matrix class.

Attributes:

- matrixState: the initial state of the problem. (inherited from the generic search problem super class)

Methods:

- **Constructor:** initializes the matrix with an initial state
- **Initial state getter and setter**
- **gengrid:** generates the string that represents the initial state of the matrix. The string is built by the following algorithm. First, a random function *randInt()* (implemented in our Helpers.java class) is used to generate values for the grid dimensions, Neo's capacity, number of hostages, number of pills, number of agents and the number of pads. A lower and upper limit is specified to *randInt()* to restrict the range of the generated random number. Once the grid dimensions and the number of objects are determined, the locations of the objects are generated. This is done by creating *Location* objects for all possible locations (according to the generated grid dimensions) and placing them in a queue. The queue is then randomly shuffled and the queue is polled one by one to determine the locations of the objects. All the information regarding the dimensions, number of objects and the location of objects is subsequently appended to the string. The string is finally returned by the method.
- **Solve**: searches for a solution for the given matrix using the indicated search strategy where the matrix and the search strategy are inputs. The method begins by creating a *MatrixState* object for the initial state. This is done by calling the method parseGrid() (implemented in our Helpers.java class) and passing the grid string to it. ParseGrid() parses the grid string and returns the initial state. The initial states is an object containing the following attributes:
    - *Location* object containing information about the grid dimensions
    - A *Neo* object representing Neo (containing attributes about his current location, damage, current capacity and original capacity)
    - An arraylist of *Hostage* objects, where each *Hostage* object contains attributes about the hostage's location, damage and whether he's being carries
    - An array list of *Location* objects, where each *Location* object represents the location of particular **alive** agent
    - A hashtable of *Location* objects, storing information about which flying pad is connected to which one.

    - An array list of *Location* objects, where each *Location* object represents the location of particular **unused** pill
    - A *Location* object representing the location of the telephone booth.

  Using the returned initial state, a *Matrix* object representing the instance of the problem that we are trying to solve is created. The *Matrix* object (representing the problem) along with the specified search strategy is passed to the *searchProcedure* method (implemented in the *Search* class). A thorough explanation of the *searchProcedure* along with the rest of the *Search* class is provided later on in the report. Thus, for now, we will give a black box explanation of the *searchProcedure* method. Once it is passed the problem and the strategy, the method returns a *Node* object representing the goal node (or null if there is no solution) and the number of expanded nodes. The output of the *searchProcedure* is then passed to the *solutionStr* method (implemented in the *Helpers* class) which traverses the path from the goal node to the root one to generate a solution string. Finally, if visualize is set to be true, the *visualize* method (also implemented in the *Helpers* class) is called to print the grid, action applied, path cost and heuristic at each depth of the path. The *visualize* method works in a recursive manner to generate its output.
- **Actions**: gives all possible actions that could be applied on a given matrix state. The method takes as input the current state. Next it initializes an empty arraylist that it will store the applicable operators in the particular state. For each of the 9 available operators **(up, down, left, right, takePill, fly, kill, carry and drop)** thorough checks via loops and if conditions are applied to check if the operation applies. If it does, the operator is placed in the arraylist. The arraylist is finally returned as the output of the method. If no action is applicable, the array list will be empty.
- **Results**: gives the result state after applying a given operator on a given matrix state. The method makes a **deep cloned copy** of the parent state and then commences to modify the attributes of the state. Generally speaking, it comprises a switch statement holding all possible actions. In each of the 9 case statements of the switch (one case statement for each action), the state is modified as

deemed appropriate. In addition to the switch statement, a for loop exists in the method. The function of the for loop is to take care of the world updates (i.e. update the damage of the alive hostages).

- **isGoal:** checks if a given state is the goal state. The state is determined to be the goal state if **all** of the following conditions are satisfied:
    - Neo's damage **is less than 100**
    - Neo's current location is the **telephone booth**
    - All of the hostages turned into agents are **killed**
    - All of the remaining agents (whether dead or alive) are **at the telephone booth and are not being carried.**
- **Stepcost**: outputs the step cost of moving from parent state s to child state s* by applying a given operation on a given matrix state. The step cost is represented as a 2-tuple (implemented in the code as 1D array of size 2). The first value in the tuple represents the number of dead hostages **(whether they turned into agents or not)** that have just died by taking the transition from state s to state s*. The second value represents the number of agents that have just been killed **(including the number of agents that were hostages before)** by taking the transition from state s to state s***.**
  Note: the **path cost** is the summation of all the **individual step cost 2-tuples** from the root to the current node. In other words, the path cost is a 2-tuple that has the following form:
  Path cost = (total number of dead hostages, total number of killed agents).
- **ASHeurastic1**: outputs a heuristic value that is driven by an admissible heuristic function. **More details on that later on in the report.**
- **ASHeurastic2**: outputs a heuristic value that is driven by an admissible heuristic function. **More details on that later on in the report.**
- **GreedHeurastic1**: outputs a heuristic value that is driven by an heuristic function. **More details on that later on in the report.**
- **GreedHeurastic2**: outputs a heuristic value that is driven by an heuristic function. **More details on that later on in the report.**

3) Implementation of the main functions

A) gengrid()

This method is responsible for generating the string that represents the grid. This method first randomly generates the dimension of the grid to be between 5*5 and 15*15. Then the method assigns a random number of the hostage capacity of neo, the number of hostages, and the number of pills. The method then calculates the possible number of agents that will be in the grid to be between 1 and N where
N = x_dimension_of_the_grid *
y_dimension_of_the_grid-2-number of hostages - number of pills.
After that the method calculates the number of possible pills to be an even number between 0 and the remaining locations in the grid. After calculating numbers of all the objects in the grid, the method constructs a queue of all possible locations in the grid where these locations are shuffled, then each object in the grid (neo, pill, hostage, etc) is assigned a location from the shuffled locations that exist in the que. This will guarantee that no two objects lie in the same location.

B) actions()

This method takes as an input the current state of the search problem node and then outputs all applicable actions(operators) for this node. The first group of actions is the ones concerned with the movement of neo, this method will assure that neo can move to a certain location if there is no wall, agent, or turned hostage in that location. The second action that could be done is carry where the method will only output that operator if there is a hostage in the same location as neo and neo can carry it and that hostage is not at the TB and the hostage is alive. The third action which is drop will only be an output of this method if Neo is carrying a hostage and he reached the TB. The forth action , Kill, will be an output of this method if there is an agent or turned hostage in an adjacent cell to neo. The fifth action which is take_pill will only be an output of this method if there is a pill in the same location as neo. The sixth and final action which is fly will only be an output of this method if there is a pad in the same location as neo.

C) results()

The input of this method is the current state of neo and an applicable action generated from the actions method and the result is the new state after applying the action.

D) stepcost()

As the cost in our implementation is the number of dead hostages and the number of killed agents(or turned hostages), this method is responsible for calculating that cost given the old state, the new state, and the applied action. The method will increment the number of dead hostages if the applied action is not taken and if there is a hostage, or number of hostages, that will die after applying this action. This method will also increment the number of killed agents or turned hostages if the action applied was killed. The method will then return the number cost of dead hostages and killed agents as a tuple.

E) Isgoal()

This method is responsible for checking if a given state is a goal state. This method returns true if Neo is alive, at the telephone booth, and all the hostages whether they died while being carried or alive are at the TB and that all hostages that turned to agents are killed.

F) solve()

The inputs of this method are the string representing the grid and a string representing the search strategy that will be used. The method first parses the string representing the grid as a matrix state by splitting the text into segments where each two segments are separated by ; in the grid string. Then the segments are converted to the actual data of the grid like the location of neo, hostages, agents, pills, pads and the maximum capacity of neo along with the health of each hostage. Having the initial state of the matrix created, the method creates a matrix problem using this matrix state and then deduces the solution of this matrix problem by utilizing the search strategy that is indicated by the second input string.

4) Implementation of the search algorithms

A *search* class has been implemented to encompass the logic of the various search strategies. Generally speaking, the class includes eight methods. Specifically, it includes a method containing the logic of the general search procedure as well as a method for each of the six different search strategies (depth first, breadth first, uniform cost, iterative deepening, greedy and A* search). The eighth method executes the logic of depth limited search. This is used as a helper method when executing iterative deepening search.

The general search procedure method, named *searchProcedure*, receives a search problem and a string, indicating the strategy, as input. The method outputs the solution, represented as an array containing the goal node and the number of expanded nodes. The method comprises a switch statements which uses the strategy string to determine which strategy method should be called and, in the case if the chosen strategy is A* or greedy, which of the two heuristics should be chosen.

Each of the six strategy methods receives the problem and the root node as input and outputs the goal state along with the number of expanded states. Below is a detailed explanation of each of the 6 strategy methods.

a) Breadth first search (BFS)

Our implementation of BFS starts with an initialization of a FIFO queue. Subsequently, the root node, that is received as input to the BFS method, is enqueued into the initialized queue. In addition to the FIFO queue, a hashset is initialized. The said hashset is used to store the set of visited states (i.e. the set of states belonging to nodes enqueued into the FIFO queue). The purpose of storing visited states in the initialized hashset is to prevent the occurrence of repeated states. Once the hashset is initialized, the state of the root node is stored in it. Finally, a variable is created with an initial value of zero to keep track of the expanded nodes.

Next the logic of the strategy encompases a loop that repeats as long as the queue is not empty. In each iteration of the loop, the front node in the FIFO queue is dequeued and the number of expanded nodes is incremented by one. Next, the goal state, implemented as a java method in the *matrix* class, is applied to the state of the dequeued node. If the state passes the goal test , then the goal node along with the number of expanded nodes is returned. Otherwise, an action method, that has been implemented in the *matrix* class , is applied to the state of the dequeued node to obtain the set of possible actions that can be applied by Neo. Each of the possible actions is subsequently applied on the dequeued state to obtain the possible next states. To avoid repeated states, as previously mentioned, the created hashset is searched to check if it already contains the state. If it does, then this state is skipped. If it doesn't, then a new node is initialized. Inside the newly initialized node, the expanded state, the state of the dequeued state and the operator used to obtain the new state are placed as the state, parent state and action, respectively. Additionally, the depth of the node is set as the depth of the parent node plus one, the pathcost is computed as the sum of the parent node's pathcost and the step cost from the parent node to the newly initialized. It should be noted that the path cost is not needed in BFS. However, it is calculated since we need to know the number of hostages dead and the number of agents killed for the output of the solve method. Finally, the heuristic value of the node is set to 0, since no heuristic will be used in BFS. Once the new node is initialized with previously mentioned data, it is enqueued at the back of the FIFO queue and its state is placed in the expanded states hashset.

The previously mentioned loop is repeated until, either a goal state is dequeued to the front of the queue and returned or the FIFO queue becomes empty. If the queue becomes empty, null is returned, indicating that no goal state exists, along with the number of expanded nodes.

b) Depth first search (DFS)

The method encompassing the logic of the DFS strategy is a mirror of the BFS one with a few minor differences. Instead of using a FIFO queue to enqueue and dequeue the nodes, a stack is used instead to push and pop them. This allows the algorithm to function in a depth first manner.

c) **Uniform Cost (UC)**
Once again, great similarities between the logic of the BFS and DFS and the logic of UC can be seen. The differences consist in the fact that, unlike in the case of BFS and DFS where a FIFO queue and stack are used, respectively, a priority queue is utilized instead. Nodes are placed in the priority queue and are sorted according to the path cost by using the *compareTo* method of the *Node* object. The *compareTo* method sorts the node by placing nodes with less hostage death (the first value in the pathcost array) at the front of the priority queue. If there is a tie in the number of hostage deaths, the node with the least number of killed agents (the second value in the pathcost array) is placed in front of the one with the larger number of kills. Once again, the heuristic value is set to 0 since no heuristics are used in uniform cost search.

d) **Iterative Deepening Search**
In the method encompassing the logic of the IDS, a for loop iterates on all possible depth values starting from depth equals to 0 until depth equals to maximum depth. In each iteration, a depth limited search (DLS) is performed by calling the helper *DLS* method and specifying the depth limit to it as input. Once the helper method performs the DLS, the number of expanded nodes of this iteration is added to the summation of expanded nodes of the previous iterations (the variable is initialized to 0 as in the previous strategies). Additionally, we check if DLS has returned the goal state by applying the goal test method on the node. If the node is the goal node, then the node along with the number of expanded states is returned. Otherwise, the subsequent iteration is executed.

Regarding the implementation of the DLS, its logic is equivalent to that of the DFS method with only two differences.

First, the depth is passed as an input as previously mentioned. Additionally, the algorithm doesn't consider nodes that are in level beyond the depth limit.

e) **Greedy**
Similar to uniform cost search, a priority queue is used to store the nodes. Once again, similar to all previously mentioned algorithms, a hashset is initialized to store the expanded states. Next, a while loop is used to continuously dequeue the front node, check if the node's state is the goal state, expand the node's children states using the set of possible actions (if the goal test fails) and create a new node for a child state (if it is not contained in the hashset). The loop is repeated until either the priority queue is empty or a goal state is reached and returned.

However, unlike uniform cost, the path cost is not used by the *compareTo* to sort the nodes in the priority queue. Instead, a heuristic value is calculated by using one of the two created greedy heuristic functions. Which of the two heuristic functions used is determined by a string passed as input to the function. This heuristic value is ten placed in the newly created node and is used by the *compareTo* method to solely determine the position of the node in the priority queue. It should be noted that the cost is still calculated and placed in the node since we need to know the number of deaths and kills. But, it's not used in the sorting of the nodes in the priority queue.

f) **A***
The A* strategy is an exact replica of the greedy approach. The only difference in the implementation comes in the fact of how the nodes are sorted in the priority queue. In the greedy algorithm, we stated that the *compareTo* method uses the heuristic to solely determine the position of the node in the priority queue. In the case of the A*, both the cost and the heuristic values are summed together and the summation is used by the *compareTo* method to determine the positions of the node in the queue. More specifically, the summation of the path cost and the heuristic value is done as

follows. The first value of the cost tuple (the number of hostages deaths) is summed with the heuristic value. If there exists a tie between the summation of the number of hostage deaths and heuristic for both nodes, then the number of agents killed is added to the sums in order to break the tie.

5) <u>Employed heuristic functions</u>

a) A* Heuristic 1

The first A* heuristic assigns a cost value that corresponds to a relaxed problem with less restrictions on the operators. This is done by ignoring the cost of movements and carry operations. The cost of movement is eliminated due to the difficulty to estimate it. The cost depends on the path that Neo chooses to take and whether Neo decides to use one or multiple pads to teleport . Consequently, in the relaxed problem, only the cost of drop, kill agents and take pills actions is considered as seen below:
h(n) = cost of drop operations + cost of kill operations + cost of take pills operations.

The cost of the drop operations is estimated to be the number of drop operations required if Neo is currently not carrying any hostage and he decides to carry the max possible hostages before dropping them at the booth. In other words it is equal to ceil(number of unsaved/maxCapacity).

The cost of the kill operation is considered to be the number of agents that Neo is required to kill (i.e those who were initially hostages) divided by 3. In case, the number of agents required to be killed is not divisible by 3, we round up the result (i.e. ceil(nKilled/3). We divide by 3 since, at best, there will be a hostage that has turned agent at 3 out of 4 of the adjacent cells. Thus, with each kill operation that is being executed, three of the required kills are accomplished simultaneously. We cannot kill 4 simultaneously because in order to reach the specified cell and execute the kill operation, you need one side to be agentless.

The number of pills taken is estimated as the number of required pills that Neo must take to stay alive. Each time Neo executes a REQUIRED kill operation, damage is increased by 20. Thus, Neo must take at least enough pills to maintain the damage below 100 after executing the mandatory kill operations. This corresponds to the following value:

H(n) = 0, if neoCurrentDamage + num_required_kills*20 <100, where *neoCurrentDamage* is the current amount of damage of Neo and *num_required_kills* is the number of kill operations that Neo will perform to kill the hostages turned into agents.

H(n) = floor([neoCurrentDamage + num_required_kills*20-100]/20) +1, if neoCurrentDamage + num_required_kills*20 >= 100.

Note: 1 is added to the overall heuristic value **in case if the heuristic value is 0 and the node is not a goal node.** This is valid because if the heuristic value is 0 and we did not reach a goal node, this means that we need at least one 1 action to reach the goal node.

-Why is it admissible?

To begin with, the actual cost of the drop operations will never be less than the estimated number of drop operations. This is due to the fact that in the estimated cost we assume that, everytime, Neo will take exactly *n* hostages at a time to the telephone booth, where *n* corresponds to the maximum capacity of Neo. Therefore, Neo will be required to take ceil(number of hostages/n) trips to drop the hostages. For the number of trips to the booth/ drop operations to be less than that, Neo would have to carry a number of hostages that exceeds his maximum capacity. However, this is not possible. Not only is the actual number of drop operations guaranteed to not be less than the estimated value, but it is also possible that it will be greater. This is because, in reality, Neo might carry less hostages at a time. As a result, Neo will be required to take more trips and will require more drop actions.

Regarding the number of kill actions, the actual cost will at least equal the estimated cost. This is due to the fact that Neo has to at least kill the hostages who have turned into agents. Additionally, at best, Neo can kill at most four at a time. This will be the case if he can stand at a cell, where there will be a single hostage turned agent at each of the adjacent cells. He cannot kill more than four because at each adjacent cell, there can be at most 1 hostage as discussed previously. Thus, the actual cost can never be less than the (number of hostages turned into agents)/4. Additionally, in reality, the actual cost can exceed the estimated cost since Neo might kill additional agents (agents that were not originally agents). Moreover, Neo might not be able to find a cell where he can eliminate four agents at a time. For the above reasons, the actual cost will either equal or exceed the estimated cost.

Finally, the actual cost of taking the pill will never be less than estimated cost since, if Neo ends up taking less pills, his damage will at some point exceed 100. Therefore, Neo will die and fail to reach the goal state.

Since each of the terms in the heuristic function is guaranteed to be less than or equal to the actual cost, the overall heuristic value must also be less than or equal to the overall actual cost. Consequently, the heuristic is admissible.

b) A* Heuristic 2

The second A* heuristic assigns a cost value that corresponds to the number of kill operations that Neo must perform. Similar to the previous A* heuristic, the cost of the kill operation is considered to be the number of agents that Neo is required to kill (i.e those who were initially hostages) divided by 3. In case, the number of agents required to be killed is not divisible by 3, we round up the result (i.e. ceil(nKilled/3). We divide by 3 since, at best, there will be a hostage that has turned agent at 3 out of 4 of the adjacent cells. Thus, with each kill operation that is being executed, three of the required kills are accomplished simultaneously. We cannot kill 4 simultaneously because in order to reach the specified cell and execute the kill operation, you need one side to be agentless. This is indicated by the function below:

h(n) = cost of kill operations = count(hostages turned agents)/3, , where *hostages turned agents* is a subset of hostages whose current location is not the telephone booth and whose damage is equal to 100.

<u>**Note:**</u> 1 is added to the overall heuristic value **in case if the heuristic value is 0 and the node is not a goal node.** This is valid because if the heuristic value is 0 and we did not reach a goal node, this means that we need at least one 1 action to reach the goal node.

-Why is it admissible?
To begin with, the total number of actions executed by Neo (the actual cost) will never be less than the estimated cost. This is due to the fact that we ignore the move, carry, fly and take pill operation that Neo will do in addition to the kill operations. We only consider the number of kill operations in the computation of the estimated cost.

Additionally, the actual number of kill operations can never be less than the estimated value. This is due to the fact that Neo has to at least kill the hostages who have turned into agents. Additionally, at best, Neo can kill at most four at a time. This will be the case if he can stand at a cell, where there will be a single hostage turned agent at each of the adjacent cells. He cannot kill more than four because at each adjacent cell, there can be at most 1 hostage as discussed previously. In reality, the actual number of kills can exceed the estimated cost since Neo might kill additional agents (agents that were not originally agents). Moreover, Neo might not be able to find a cell where he can eliminate four agents at a time. Consequently, for the two above reasons, the actual cost will either equal or exceed the estimated cost and ,thus, this heuristic is admissible.

c) **Greedy Heuristic 1**
As in A* heuristic 2, the first greedy heuristic assigns to each node a cost value that is equivalent to the estimated cost of required kills. The estimation of the required kills is computed in the same way as in the A* heuristic.

<u>**Note:**</u> 1 is added to the overall heuristic value **in case if the heuristic value is 0 and the node is not a goal node.** This is

valid because if the heuristic value is 0 and we did not reach a goal node, this means that we need at least one 1 action to reach the goal node.

d) **Greedy Heuristic 2**
The second greedy heuristic assigns a cost value that corresponds to the cost of “take pills” actions. The number of “take pills” actions is estimated to be the number of required pills that Neo must take to stay alive. Each time Neo executes a REQUIRED kill operation, damage is increased by 20. Thus, Neo must take at least enough pills to maintain the damage below 100 after executing the mandatory kill operation. This corresponds to the following value:

H(n) = 0, if neoCurrentDamage + num_required_kills*20 <100, where *neoCurrentDamage* is the current amount of damage of Neo and *num_required_kills* is the number of kill operations that Neo will perform to kill the hostages turned into agents.

H(n) = floor([neoCurrentDamage + num_required_kills*20-100]/20) +1, if neoCurrentDamage + num_required_kills*20 >= 100.

<u>**Note:**</u> 1 is added to the overall heuristic value **in case if the heuristic value is 0 and the node is not a goal node.** This is valid because if the heuristic value is 0 and we did not reach a goal node, this means that we need at least one 1 action to reach the goal node.

6) <u>Two running examples of the implementation</u>

a) Example 1

<u>**Input**</u>
Grid: 5,5;2;3,4;1,2;0,3,1,4;2,3;4,4,0,2,0,2,4,4;2,2,91,2,4,62
Strategy: BF (Breadth first search)
Visualize: True

<u>**Output**</u>

```
=======================NODE-AT-DEPTH-0=======================
ACTION: NOP, PATH COST: [0, 0], HEURISTIC: 0.0
```

```
   |   0   |   1  |   2   |   3   |   4   |
---+-------+-------+-------+-------+-------+
 0 |       |       |P(4,4);|  A;   |       |
----+-------+-------+-------+-------+-------+
 1 |       |       |  TB;  |       |  A;   |
----+-------+-------+-------+-------+-------+
 2 |       |       |H(91); |  P;   |H(62); |
----+-------+-------+-------+-------+-------+
 3 |       |       |       |       |N(0,2);|
----+-------+-------+-------+-------+-------+
 4 |       |       |       |       |P(0,2);|
```

```
=======================NODE-AT-DEPTH-1=======================
ACTION: UP, PATH COST: [0, 0], HEURISTIC: 0.0

    |     0      |     1      |     2      |     3      |     4      |
----+------------+------------+------------+------------+------------+
 0 |            |            |  P(4,4);   |    A;      |            |
----+------------+------------+------------+------------+------------+
 1 |            |            |    TB;     |            |    A;      |
----+------------+------------+------------+------------+------------+
 2 |            |            |  H(93);    |    P;      |N(0,2);H(64);|
----+------------+------------+------------+------------+------------+
 3 |            |            |            |            |            |
----+------------+------------+------------+------------+------------+
 4 |            |            |            |            |  P(0,2);   |
```

```
=======================NODE-AT-DEPTH-2=======================
ACTION: CARRY, PATH COST: [0, 0], HEURISTIC: 0.0

   |        0        |        1        |        2        |        3        |        4        |
---+------------------+------------------+-------------------+------------------+------------------+
  0 |                  |                  |      P(4,4);      |         A;       |                  |
----+------------------+------------------+-------------------+------------------+------------------+
  1 |                  |                  |        TB;        |                  |         A;       |
----+------------------+-----------------+--------------------+------------------+------------------+
  2 |                  |                 |       H(95);       |         P;       |N(0,1);H(66,
CARRIED);|
----+------------------+-----------------+--------------------+------------------+------------------+
  3 |                  |                  |                   |                  |                  |
----+------------------+-----------------+--------------------+------------------+------------------+
  4 |                  |                 |                    |                   |      P(0,2);    |
```

```
=======================NODE-AT-DEPTH-3========================
ACTION: LEFT, PATH COST: [0, 0], HEURISTIC: 0.0

   |          0          |          1          |          2          |          3          |          4          |
---+--------------------+--------------------+--------------------+-------------------+--------------------+
  0 |                  |                   |      P(4,4);      |         A;        |                   |
----+-------------------+--------------------+--------------------+--------------------+-------------------+
  1 |                  |                   |        TB;        |                   |         A;        |
----+-------------------+--------------------+--------------------+--------------------+-------------------+
  2 |                  |                   |       H(97);      |N(0,1);H(68, CARRIED);P;|      |
----+-------------------+--------------------+--------------------+--------------------+-------------------+
  3 |                  |                    |                   |                    |                   |
----+-------------------+--------------------+--------------------+--------------------+-------------------+
  4 |                  |                    |                   |                    |      P(0,2);
|

=======================NODE-AT-DEPTH-4========================
ACTION: LEFT, PATH COST: [0, 0], HEURISTIC: 0.0

   |          0          |          1          |          2          |          3          |     4
|
---+----------------------+----------------------+----------------------+----------------------+---------
--+
  0 |                     |                     |        P(4,4);        |           A;           |
|
----+---------------------+----------------------+----------------------+---------------------+---------
--+
  1 |                     |                     |          TB;           |                      |     A;
|
----+---------------------+----------------------+----------------------+---------------------+---------
--+
  2 |                     |                     |N(0,1);H(99);H(70, CARRIED);|  P;   |
|
----+---------------------+----------------------+----------------------+---------------------+---------
--+
  3 |                     |                     |                       |                     |
|
----+---------------------+---------------------+---------------------+----------------------+---------
--+
  4 |                     |                     |                      |                     |
P(0,2);  |
```

```
=======================NODE-AT-DEPTH-5=======================
ACTION: UP, PATH COST: [1, 0], HEURISTIC: 0.0

   |          0          |          1          |          2          |          3          |          4
|
----+-------------------+-------------------+--------------------+--------------------+------------------
--+
  0 |                    |                    |       P(4,4);       |         A;          |
|
----+-------------------+------------------+---------------------+--------------------+------------------
--+
  1 |                    |                  |N(0,1);H(72, CARRIED);TB;|       |          A;
|
----+--------------------+-------------------+----------------------+---------------------+------------------
--+
  2 |                    |                   |       H(101);        |         P;          |
|
----+--------------------+-------------------+----------------------+---------------------+------------------
--+
  3 |                    |                   |                      |                     |
|
----+--------------------+--------------------+---------------------+----------------------+-----------------
--+
  4 |                    |                    |                     |                      |     P(0,2);
|

=======================NODE-AT-DEPTH-6========================
ACTION: KILL, PATH COST: [1, 1], HEURISTIC: 0.0

   |          0          |          1          |          2          |          3          |       4
|
----+-------------------+---------------------+--------------------+---------------------+---------------
--+
  0 |                    |                    |       P(4,4);       |          A;         |
|
----+-------------------+---------------------+---------------------+--------------------+---------------
--+
  1 |                    |                    |N(20,1);H(74, CARRIED);TB;|      |          A;
|
----+-------------------+---------------------+--------------------+---------------------+---------------
--+
  2 |                    |                    |                    |         P;          |
|
----+-------------------+---------------------+--------------------+--------------------+----------------
--+
  3 |                    |                    |                    |                    |
|
----+-------------------+--------------------+---------------------+--------------------+----------------
--+
  4 |                    |                    |                    |                    |     P(0,2);
|
=======================NODE-AT-DEPTH-7========================
ACTION: DROP, PATH COST: [1, 1], HEURISTIC: 0.0

   |       0      |       1       |        2       |        3       |        4       |
----+-------------+--------------+--------------+--------------+--------------+
  0 |             |             |   P(4,4);    |      A;       |             |
----+-------------+-------------+-----------------+--------------+-------------+
  1 |             |             |N(20,2);H(74);TB;|             |      A;     |
----+-------------+-------------+-----------------+--------------+-------------+
  2 |             |             |               |      P;       |             |
----+-------------+-------------+-----------------+--------------+-------------+
  3 |             |             |               |              |             |
----+-------------+-------------+----------------+---------------+-------------+
  4 |             |             |               |              |   P(0,2); |
```

up,carry,left,left,up,kill,drop;1;1;338

<u>Input</u>
Grid: 5,5;2;3,4;1,2;0,3,1,4;2,3;4,4,0,2,0,2,4,4;2,2,91,2,4,62
Strategy: DF (Depth first search)
Visualize: False

<u>Output</u>
up,kill,carry,up,up,kill,down,down,down,down,fly,down,drop,kill;1;3;15

<u>Input</u>
Grid: 5,5;2;3,4;1,2;0,3,1,4;2,3;4,4,0,2,0,2,4,4;2,2,91,2,4,62
Strategy: UC(Uniform cost search)
Visualize: False

<u>Output</u>
up,carry,left,takePill,left,carry,up,drop;0;0;62

__Input__
Grid: 5,5;2;3,4;1,2;0,3,1,4;2,3;4,4,0,2,0,2,4,4;2,2,91,2,4,62
Strategy: ID (Iterative deepening search)
Visualize: False

__Output__
up,carry,left,left,carry,up,drop;1;0;626

__Input__
Grid: 5,5;2;3,4;1,2;0,3,1,4;2,3;4,4,0,2,0,2,4,4;2,2,91,2,4,62
Strategy: GR1 (Greedy search heuristic 1)
Visualize: False

__Output__
up,kill,carry,up,up,kill,down,down,down,down,fly,down,drop,kill;1;3;29

__Input__
Grid: 5,5;2;3,4;1,2;0,3,1,4;2,3;4,4,0,2,0,2,4,4;2,2,91,2,4,62
Strategy: GR2(Greedy search heuristic 2)
Visualize: False

__Output__
up,kill,carry,up,up,kill,down,down,down,down,fly,down,drop,kill;1;3;29

__Input__
Grid: 5,5;2;3,4;1,2;0,3,1,4;2,3;4,4,0,2,0,2,4,4;2,2,91,2,4,62
Strategy: AS1 (A* search heuristic 1)
Visualize: False

__Output__
up,carry,left,takePill,left,carry,up,drop;0;0;61

__Input__
Grid:_5,5;2;3,4;1,2;0,3,1,4;2,3;4,4,0,2,0,2,4,4;2,2,91,2,4,62
Strategy: AS2 (A* heuristic 2)
Visualize: False

__Output__
up,carry,left,takePill,left,carry,up,drop;0;0;61

b) Example 2

__Input__
Grid:_5,5;1;1,4;1,0;0,4;0,0,2,2;3,4,4,2,4,2,3,4;0,2,32,0,1,38
Strategy: BF (Breadth first search)
Visualize: False

__Output__
left,left,up,carry,left,left,down,drop,right,up,carry,left,down,drop;0;0;884

__Input__
Grid: 5,5;1;1,4;1,0;0,4;0,0,2,2;3,4,4,2,4,2,3,4;0,2,32,0,1,38
Strategy: DF (Depth first search)
Visualize: False

__Output__
kill,up,left,left,carry,down,down,takePill,up,up,left,left,takePill,down,drop,up,right,carry,down,left,drop;0;1;41

__Input__
Grid: 5,5;1;1,4;1,0;0,4;0,0,2,2;3,4,4,2,4,2,3,4;0,2,32,0,1,38
Strategy: UC(Uniform cost search)
Visualize: False

__Output__
left,up,left,left,left,takePill,right,carry,down,left,drop,up,right,right,carry,down,down,takePill,up,up,left,left,down,drop;0;0;126

Input
Grid: 5,5;1;1,4;1,0;0,4;0,0,2,2;3,4,4,2,4,2,3,4;0,2,32,0,1,38
Strategy: ID (Iterative deepening search)
Visualize: False

Output
kill,up,left,left,left,carry,down,left,drop,up,right,right,carry,down,left,left,drop;0;1;3647

Input
Grid: 5,5;1;1,4;1,0;0,4;0,0,2,2;3,4,4,2,4,2,3,4;0,2,32,0,1,38
Strategy: GR1 (Greedy search heuristic 1)
Visualize: False

Output
left,up,left,carry,down,down,takePill,up,up,left,left,takePill,down,drop,up,right,carry,down,down,down,down,right,fly,up,up,kill,up,left,left,down,down,left,left,up,drop;0;1;74

Input
Grid: 5,5;1;1,4;1,0;0,4;0,0,2,2;3,4,4,2,4,2,3,4;0,2,32,0,1,38
Strategy: GR2(Greedy search heuristic 2)
Visualize: False

Output
left,up,left,carry,down,down,takePill,up,up,left,left,takePill,down,drop,up,right,carry,down,down,down,down,right,fly,up,up,kill,up,left,left,down,down,left,left,up,drop;0;1;74

Input
Grid: 5,5;1;1,4;1,0;0,4;0,0,2,2;3,4,4,2,4,2,3,4;0,2,32,0,1,38
Strategy: AS1 (A* search heuristic 1)
Visualize: False

Output
left,up,left,left,left,takePill,right,carry,down,left,drop,up,right,right,carry,down,down,takePill,up,up,left,left,down,drop;0;0;125

Input
Grid: 5,5;1;1,4;1,0;0,4;0,0,2,2;3,4,4,2,4,2,3,4;0,2,32,0,1,38
Strategy: AS2 (A* heuristic 2)
Visualize: False

Output
left,up,left,left,left,takePill,right,carry,down,left,drop,up,right,right,carry,down,down,takePill,up,up,left,left,down,drop;0;0;125

7) Comparison of the performance of the implemented search strategies

a) Example 1

	Completeness	Optimality	RAM Usage	CPU Utilization	Expanded Nodes
BF	Yes	No	242.2 MB	3.7%	338
DF	Yes	No	243.3 MB	5.3%	15
UC	Yes	Yes	244.8 MB	5.5%	62
ID	Yes	No	243.8 B	5.5%	626
GR1	Yes	No	238 MB	5.3%	29
GR2	Yes	No	238.2 MB	6.3%	29
AS1	Yes	Yes	238.2 MB	5.3%	61
AS2	Yes	Yes	238.3 MB	6.3%	61

NOTE: although DFS is generally not complete, it will be complete in our case since we handle repeated states. Thus, we can never go down a particular depth indefinitely.

b) Example 2

	Completeness	Optimality	RAM Usage	CPU Utilization	Expanded Nodes
BF	Yes	Yes	250.4 MB	4.7%	884
DF	Yes	No	250.5 MB	3.5%	41
UC	Yes	Yes	250.8 MB	4.3%	126
ID	Yes	No	250.9 B	4.5%	3647
GR1	Yes	No	249.9 MB	3.6%	74
GR2	Yes	No	249.7 MB	5.0%	74
AS1	Yes	Yes	250.9 MB	6.6%	125
AS2	Yes	Yes	251.4 MB	6.4%	125

NOTE: although DFS is generally not complete, it will be complete in our case since we handle repeated states. Thus, we can never go down a particular depth indefinitely.

C)Performance comparison and comments:
The tables and the input output examples show that the A* search strategy is the best strategy as it is optimally efficient, meaning it is guaranteed to give the optimal solution while expanding the least number of nodes. The tables and the input output examples also show that uniform cost strategy also gives the optimal solution, yet, it expands more nodes than the A* strategy. As for the rest of the search strategies, the tables and the input output examples show that most of them expand more number of nodes than the A* and most of the times they expand less number of nodes than the Uniform cost, yet, there is no guarantee

that any of them will reach the optimal solution as they will just return a goal node if there is one.

Regarding the CPU usage, the tables show that as expected the search strategies that reorder the nodes in the queue based on their path cost or heuristic value or both (A*, Greedy, UC) utilize the CPU the most. This is because every time a node must be added to the queue, a value comparison between this node and all other nodes in the queue has to be carried over to put this new node in its right place. As for the Iterative deepening search, the CPU utilization is also high due to employing depth limited search to a number of times that is equal to the max depth identified by the IDS strategy. For the BFS, DFS, the CPU utilization is less as the new nodes get added to the queue in the front (DFS) or at the back (BFS).

For the RAM utilization, the tables prove that the BFS, DFS, IDS will have the highest space complexity; this is because the maximum number of nodes that any of these strategies will need to keep track of is O(B^D). For UC search the tables also indicate high RAM usage as the space complexity of the UCis also O(B^D). For the greedy search, the RAM utilized is less than BFS, DFS, IDS, and UC although the worst theoretical space complexity is O(B^D) which indicates the quality of the heuristics employed. For the A* search, the RAM utilization is close to the Utilization of the greedy although the worst theoretical space complexity is O(B^D) which also indicates the quality of the admissible heuristic functions.

8) <u>Citations</u>

- Stuart Russel and Peter Norvig (2003). *Artificial Intelligence: A Modern Approach (2nd edition).* Prentice Hall. ISBN 0137903952.