AI course - Assignment 2

Amirali Rayegan, 200538720

Question 1

In the first step, the code reads the input files line by line. The heuristic of each node will be saved to a dictionary, which is called ***heuristics,*** in the following format:
[‘s’:45, ‘A’:30, ……. , ‘G1’:0, ‘G2’:0]

Also, the routes between each two nodes will be saved in a dictionary called ***map***, with the following format:
[[‘S’, ‘A’]: 15, [‘A’, ‘B’]:2, ……. , [‘F’, ‘G2’]:70, [‘G1’,’G2’]:10]

Then, I created a list that works as a fringe for the A* algorithm. This list is called ***route***, and at first, the only element of it is the root. Each element of this list follows this format:
[Node, heuristic, cost]
The first element is:
[‘S’, 45, 0]

Then, there is a while that the main A* search takes place there. In that loop, first, I will look for the next node to take out of the fringe. For this purpose, I check each element of the route and find the one with the lowest “Heuristic + Cost.” Then, In the next lines, I check whether this element is the goal or not. In case it is the goal, I use the break command to get out of the loop.

At the next step, I look forward to finding the potential next nodes. In this regard, I created a list called ***possiblePath*** and added the potential next nodes.

Next, I will add all of the elements of the ***possiblePath*** to ***route***. The mentioned task is performed in the following format:
route at first: [[‘S,’ 45, 0]]
route after first iteration: [[‘SA,’ 30, 15] , [‘SB,’ 40, 2]]

This whole process repeats until the chosen node is a goal.

Question 2

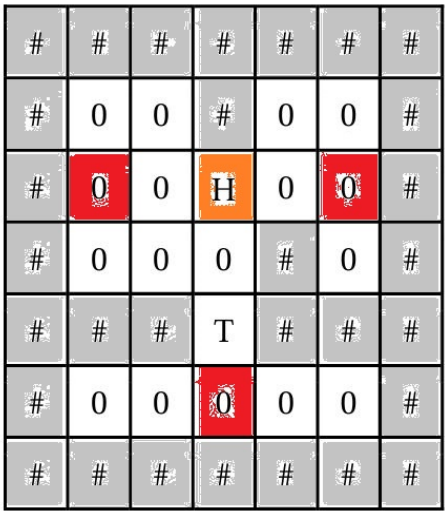
Overall, the main part of the question 2 solutions takes place in 3 steps. First, read the board and find Harry on it. Second, the desired algorithm and call-related function are defined. Lastly, regarding the result, a suitable output will be generated. It is shown as follows:

```
## Getting Input from the input file
# Game board
board = readBoardFromFile(sys.argv[1], sys.argv[2])

# Harry's location
harryRow, harryColumn = findHarry(board)

# A boolean that identifies the possibility of finding a path
isImpossible = False

# Performing appropriate algorithm
algorithm = sys.argv[2]
if (algorithm == 'dfs'):
    isImpossible = dfs(harryRow, harryColumn, board, [])
elif (algorithm == 'bfs'):
    isImpossible = bfs(harryRow, harryColumn, board)
elif (algorithm == 'astar'):
    isImpossible = astar(harryRow, harryColumn, board)

# Printing output if there is no path
if isImpossible is False and algorithm != 'astar':
    printImpossible(algorithm)
```

The ***readBoardFromFile*** and ***findHarry*** are two simple functions that go through the input/board. The format of the parameters and the returning value of each function are available in the comments. The ***printImpossible*** function is used for DFS and BFS algorithms, creating a text file containing the words ‘Not possible.’

DFS

The main functionality of the DFS algorithm is implemented in the ***dfs*** function. Regarding the fact that this function is recursive, it is important to note the inputs and returning values of this function. The parameters of this function are the row and column of the current node, the whole board, and a list that indicates the nodes that have been previously visited. It returns True if that branch, starting with ***startRow*** and ***startColumn***, reaches the goal and False if not. At the start of this function, it checks whether the passing node (***startRow*** and ***startColumn***) is in the visited list or not. If it is, then it returns False. If not, it adds the node to the visited list and continues to the following lines. Then, it checks If the current node is the goal or not. If it is, the function for writing the result path in an output file will be called, and True will be returned.
In the next step, the next possible tiles should be discovered. In this regard, two new variables called ***newRow*** and ***newColumn*** are used. These two variables represent the location of the next-most left, right, up, and down tile, respectively. For example, if we are on the H (the orange node), then the next possible nodes are the red ones.



Here is the pseudo-code for the next step:

1. Find the next possible note
2. Check whether it is already on the visited list or not
3. Call the ***dfs*** function starting with this point
4. Return True if it returns True
5. Otherwise, go to step 1.

These five steps are repeated four times (per each next possible node: left, right, up, down). If none of them returns True, then it means there is no path to the destination in all branches, and it returns false.
More detailed information about the implementation of this algorithm is available in the comments.

BFS

Implementing this algorithm is so similar to the A* implementation. There is a list called ***fringe,*** which keeps the nodes that are going to be explored next. The only difference is that in each step, we do not search for the lowest (heuristic + cost) and simply choose the first node.
In parallel with ***fringe***, we have another list that keeps track of the path from the starting node to the current node. These two lists follow this format:
fringe: [[2,3] , [2,1]]
path: [[[1,2] , [2,2] , [2,3]] , [[1,2] , [2,2] , [2,1]]]
Each time an element is removed from the ***fringe*** or added to the fringe the same thing will apply to the ***path***.

Here are the steps in the main loop:

1. Pick up the first element of the ***fringe***
2. If it is a goal, call the function for generating the result in the output file
3. Add that node to the ***visited*** list
4. Find a potential next node
5. Check whether it has already been visited
6. If not, add it to the end of the ***fringe*** and create a new path to add it to the end of the ***path***
7. Repeat steps 4-6 four times in each iteration (Left, right, up, down)
8. Go to step 1

If there is no element left in the ***fringe,*** it means the agent could not find a possible path, and the bfs function returns False.
More detailed information about the implementation of this algorithm is available in the comments.

A*

Although there are a few differences, the implementation of this part is similar to question one since they both use the same algorithm. The ***astar*** function starts by checking whether the destination is in the right location (the most right-down tile). After that, it calls the ***findKeys*** function to find the location of keys on the board. Here, there is a list called ***keys***, which keeps the attributes of all keys on the board. Each key is a list with three variables as follows:
key: [row, column, the order in which this key is visited (0 by default)]
For example, if the agent goes to the key in the (1,2) position first and then visits a key in the (4,5) position and also there is one other key to explore in the (7,1) position, then the ***keys*** list is formatted as follows:
keys: [[1,2,1] , [4,5,2] , [7,1,0]]
The purpose of the third element is to define the order of visited keys when generating output. After that, there is a list called ***fringe*** which works the same as the ***route*** in question one. The format of each element in the ***fringe*** is like this:
Each node: [Node position, Noe Heuristic, Cost from start to this node, list of keys]
In parallel with ***fringe,*** there is another list called ***routes*** that keeps track of the route each node passes. Per each node in the ***fringe***, there is a path in ***routes***.

Then, we have a list of visited nodes that is called ***visited***.

Then, the main loop of the A* search algorithm starts and the following tasks take place in each iteration:

1. Looking into the fringe to find the next node (Lowest cost + Heuristic)
2. Pick up that node and update ***fringe***, ***route***, and ***visited***
3. Check if we reach the goal, and if we do, call the function ***printOutputAStar*** to write the desired output into the file.
4. Check if we reach any key, and if we do, change the third element of that key in the ***keys***
5. Create a list called ***possiblePath*** and add four nodes to it. These four nodes represent the position of right tile, down tile, left tile, and up tile, respectively.
6. Per each node in ***possiblePath*** check whether that location is permitted to go on it on the board or not.
7. If yes, calculate the cost of going to that node (current cost + 1) and the heuristic of that node using the ***findHeuristic*** function. ***findHeuristic*** is a function that gets the current node location, the current list of keys, and the destination location and returns an integer which is the estimation of the distance between the current node and destination.
8. Then, using the heuristic and cost of each proposed node, we will check whether there is a node with the current node's position and a lower (cost + Heuristic) in the ***fringe*** list and the ***visited*** list or not.
9. If there is an element in one of those lists with the mentioned characteristic, we ignore that node, and if there is not, we will update ***fringe*** and ***routes***.
10. Steps 6-9 are repeated for each proposed node (four times).
11. If after all that procedure fringe is empty, then there is no possible path and the function calls the ***printImpossibleAStar*** function to write appropriate output in the file.

Heuristic

Basically, the most important part of implementing the A* algorithm is designing an efficient heuristic. In this code, I used ***findHeuristic*** to find a heuristic value per node. Since the agent knows the location of each key and also the location of the destination node, I decided to choose a heuristic that is calculated with the following formula.

> For node A we have:
> H(A) = MD(A, closest key) + MD(closest key, second closest key) + MD(second closest key, third closest key) + …… + (farest key, destination)
> *MD = Manhattan Distance

This heuristic is consistent and admissible since there are walls on the map and the actual path for collecting all keys and getting to the destination is always greater than this heuristic. If there are no walls in the map, then the suggested heuristic is always equal to the shortest path.

In order to calculate the mentioned formula for each node, I implemented the ***findHeuristic*** function recursively. The pseudo-code of this function is as follows:

```
def findHeuristic(node, keys, dest):
    if all keys are visited:
        return manhattanDistance(node, dest)
    else:
        Find the closest key K
        Updating the keys list
        return manhattanDistance(node, K) +
                findHeuristic(K, updatedKeys, dest)
```

Number of expanded Paths

	A*	DFS	BFS
Maze 1	40	10	12
Maze 2	43	15	14
Maze 3	10	8	7
Maze 4	63	12	11
Maze 5	5889	29	17

The results of this table show that although the A* algorithm is optimal and complete, the resources needed to run the algorithm grow exponentially, which makes it less efficient when using a big dataset. On the other hand, BFS outperforms other algorithms when it comes to efficiency. DFS is almost similar to BFS in terms of efficiency, but it seems to need more resources than BFS when dealing with big datasets.