ASSIGNMENT 1
AI & ML

Dr. Marwan Torki

Asmaa Gamal
Communication & Electronics

Using Informed and Uninformed Search Algorithms To Solve 8-puzzle

The Code

1.The “main.py”:

```
# Asmaa Gamal
#Assignment 1
# Electronics & Communications department
#using informed and uninfoormed search Algorithms to solve 8-puzzle


#------------------------------------------libiraries-----------------------------------------------
import  math
import  timeit                                                                         #Measure execution
time of small code
from    heapq            import heappush, heappop, heapify
from    ClassOfStates    import State                                                  #importing from
class
from    collections      import deque
from    GUI              import  GUI,print_cyan



# ----------------------------------------The 8-Puzzle Game ------------------------
print(" ---------------------8-puzzle Game using (BFS, DFS,& A*) search algorithms---------------
----------------- ")
print("-------------------------------------Welcome to the game------------------------------------
-----------------")
#initialization
goal_state    = [0, 1, 2, 3, 4, 5, 6, 7, 8]
goal_node     = State #Goal node is a full class which has depth, state,key,total cost,...  etc
initial_state = list()

#zero_initialization
board_len      = 0
board_side     = 0
nodes_expanded = 0

moves = list()  #there is another global moves in the export fun below
costs = set()


#--------------------------------------------BFS------------------------------------------------
def bfs(start_state):                        #passing the initial_state_to_BFS_algorism

    global  goal_node

    explored= set() ; queue = deque([State(start_state, None, None, 0, 0, 0)]) #queue=frontier
#q=deque ([])

    while queue: #frontier is not empty

        node = queue.popleft() #node= state object #remove and return the first in or the most
left element

        explored.add(node.map)

        if node.state == goal_state: #node=State object  # accessing the state in the class to
compare it with the goal if success
            goal_node = node
            return queue

        neighbors = expand(node)

        for neighbor in neighbors:
            if neighbor.map not in explored:
                queue.append(neighbor) #queue==frontier
```

```
                explored.add(neighbor.map)



#-------------------------------------------DFS------------------------------------------------

def dfs(start_state):

    global goal_node

    explored=set(); stack =  list([State(start_state, None, None, 0, 0, 0)]) #here this means:
list = stack, queue =deque

    while stack:                                                              #frontier = stack
is not empty

        node = stack.pop()

        explored.add(node.map)                                                #node= state object

        if node.state == goal_state:
            goal_node = node
            return stack

        neighbors = reversed(expand(node))                                   #because it's a
stack

        for neighbor in neighbors:
            if neighbor.map not in explored:
                stack.append(neighbor)
                explored.add(neighbor.map)


#----------------------------------------------A*-------------------------------------------------

#-------------------------------------------heuristics -----------------------------------------

#------------------------Manhattan_distance
def h(state):

    return sum(    abs(b % board_side - g % board_side) + abs(b//board_side - g//board_side)
               for b, g in (   ( state.index(i), goal_state.index(i)   ) for i in range(1,
board_len)  )      ) #board_len=9


#--------------------------Euclidean_distance
def eclidean_h(state):


    ecl_sum = 0
    for i in range(1, 9):
        brd= state.index(i)  ; gl = goal_state.index(i)
        x = (brd % 3 - gl % 3) ** 2
        y = (brd // 3 - gl // 3) ** 2
        ecl_sum += math.sqrt(x+y)
    return ecl_sum
'''
    # the above code lines in this fun are the same as saying:
    return sum(    math.sqrt(   ((b % board_side - g % board_side)**2 + (b//board_side -
g//board_side)**2)     )  #board_side=3
               for b, g in (   ( state.index(i), goal_state.index(i)   ) for i in range(1,
board_len)  )      )  #board_len=9
'''
#--------------------------------------A* algorithm code --------------------------------
#--------------------------------using Manhattan_distance-----------------------------
def ast(start_state):
    print('------------------------A* Using Manhattan Distance Heuristics------------------')
    global  goal_node
```

```
    explored=set(); heap=list(); heap_entry={}                      #dic #frontier=heap #not used in
any thing below  counter = itertools.count()

    key = h(start_state)                                            #heuristics manhattan distance

    root = State(start_state, None, None, 0, 0, key)                #root=neighbor

    entry = (key, 0, root)                                          #tuple does not change

    heappush(heap, entry)

    heap_entry[root.map] = entry

    while heap:

        node = heappop(heap)

        explored.add(node[2].map)

        if node[2].state == goal_state:
            goal_node = node[2]
            return heap

        neighbors = expand(node[2])

        for neighbor in neighbors:

            neighbor.key = neighbor.cost + h(neighbor.state)

            entry = (neighbor.key, neighbor.move, neighbor)             #neighbor.move means the
element neighbor in the list move

            if neighbor.map not in explored:

                heappush(heap, entry)

                explored.add(neighbor.map)

                heap_entry[neighbor.map] = entry

            elif neighbor.map in heap_entry and neighbor.key < heap_entry[neighbor.map][2].key:

                hindex = heap.index((heap_entry[neighbor.map][2].key,
                                     heap_entry[neighbor.map][2].move,
                                     heap_entry[neighbor.map][2]))

                heap[int(hindex)] = entry

                heap_entry[neighbor.map] = entry

                heapify(heap)                                           #trannsform a list into a
heap

#----------------------------------------------A* algorithm code ---------------------------------
#------------------------------------------using Euclidean_distance------------------------------
def euclidean_ast(start_state):

    global  goal_node
    explored=set(); heap=list(); heap_entry={}                          #dic #frontier=heap #not used
in any thing below  counter = itertools.count()
    key = eclidean_h(start_state)                                       #heuristics manhattan
distance

    root = State(start_state, None, None, 0, 0, key)                    #root=neighbor

    entry = (key, 0, root)  #tuple does not change

    heappush(heap, entry)
```

```python
    heap_entry[root.map] = entry

    while heap:

        node = heappop(heap)

        explored.add(node[2].map)

        if node[2].state == goal_state:
            goal_node = node[2]
            return heap

        neighbors = expand(node[2])

        for neighbor in neighbors:

            neighbor.key = neighbor.cost + eclidean_h(neighbor.state)

            entry = (neighbor.key, neighbor.move, neighbor)                 #neighbor.move means
the element neighbor in the list move

            if neighbor.map not in explored:

                heappush(heap, entry)

                explored.add(neighbor.map)

                heap_entry[neighbor.map] = entry

            elif neighbor.map in heap_entry and neighbor.key < heap_entry[neighbor.map][2].key:

                hindex = heap.index((heap_entry[neighbor.map][2].key,
                                     heap_entry[neighbor.map][2].move,
                                     heap_entry[neighbor.map][2]))

                heap[int(hindex)] = entry

                heap_entry[neighbor.map] = entry

                heapify(heap) #trannsform a list into a heap


#-------------------------------------Actions & paths ------------------------------------------
def move(state, position):

    new_state = state[:]                                                                #every
element in the array

    index = new_state.index(0)

    if position == 1:  # Up

        if index not in range(0, board_side):
#impossible = [0,1,2]

            temp = new_state[index - board_side]
#board_side=3
            new_state[index - board_side] = new_state[index]
            new_state[index] = temp

            return new_state
        else:
            return None

    if position == 2:  # Down

        if index not in range(board_len - board_side, board_len):
```

```
#impossible=range(9-3,9)=[6,7,8]

            temp = new_state[index + board_side]
            new_state[index + board_side] = new_state[index]
            new_state[index] = temp

            return new_state
        else:
            return None

    if position == 3:  # Left

        if index not in range(0, board_len, board_side):

            temp = new_state[index - 1]
            new_state[index - 1] = new_state[index]
            new_state[index] = temp

            return new_state
        else:
            return None

    if position == 4:  # Right

        if index not in range(board_side - 1, board_len, board_side):

            temp = new_state[index + 1]
            new_state[index + 1] = new_state[index]
            new_state[index] = temp

            return new_state
        else:
            return None

#--------------------------------Expanding neibours & frontiers ---------------------------------
def expand(node):

    global nodes_expanded
    nodes_expanded += 1

    neighbors = list()

    neighbors.append(State(move(node.state, 1), node, 1, node.depth + 1, node.cost + 1, 0))  #up
# state, parent, move, depth, cost, key
    neighbors.append(State(move(node.state, 2), node, 2, node.depth + 1, node.cost + 1, 0))
#down
    neighbors.append(State(move(node.state, 3), node, 3, node.depth + 1, node.cost + 1, 0))
#left
    neighbors.append(State(move(node.state, 4), node, 4, node.depth + 1, node.cost + 1, 0))
#right

    nodes = [neighbor for neighbor in neighbors if neighbor.state]

    return nodes


#----------------------------- preparing of the output stage-------------------------------------
def backtrace():

    current_node = goal_node

    while initial_state != current_node.state:

        if current_node.move == 1:
            movement = 'Up'
        elif current_node.move == 2:
            movement = 'Down'
        elif current_node.move == 3:
```

```
            movement = 'Left'
        else:
            movement = 'Right'

        moves.insert(0, movement)
        current_node = current_node.parent                                             #the
parent node will be considered as a currnt node for the comming check or iteration

    return moves

#----------------------------------------the output stage ---------------------------------------
def export( time ):

    global moves
    global nodes_expanded
    moves = backtrace()

    print('1.Path To Goal:  ',end = '' )
    print_cyan(str(moves))

    print('2.Cost Of Path:  ' , end = '')
    print_cyan(str(len(moves)))
    print('3.Nodes Expanded:', end = '')
    print_cyan(str(nodes_expanded))

    print('4.Search Depth:  ' , end = '')               #search depth = max search depth i found
in it my goal
    print_cyan(str(goal_node.depth))

    print('5.Running Time:  ', end = '' )
    print_cyan( format( time , '.8f'))

    # clearing everything in case we call the export method again in the A* algorithm using the
Euclidean distance
    moves = []
    nodes_expanded=0
    goal_node.depth = {}

#----------------------populating and reading  i/p from the the input stage ----------------------
def inputread(configuration):                          # populates a configuration thing

    global board_len, board_side

    data = configuration.split(",")

    for element in data:
        initial_state.append(int(element))      #adding elements

    board_len = len(initial_state)

    board_side = int(board_len ** 0.5)          #sqrt root of 9 =3



#---------------------------------------the main -------------------------------------------------
def main(algorithm,board):

    inputread(board)                                  #calling the read fun or method

    if  algor_type == 'A*' or algor_type == 'a*' or algor_type == 'a star'or algor_type == 'A
STAR'or algor_type == 'A Star'or algor_type == 'A star'or algor_type == 'A* ':
        print('-------------------------A* Using Euclidean Distance Heuristics------------------')

        start = timeit.default_timer()           # This will return the default time before
executing the next line
        euclidean_ast(initial_state)
        stop = timeit.default_timer()            # This will return the default time after
executing the above previous line
```

```
        export( stop-start)

    function = function_map[algorithm]          #google told me that this tricky line we use when we wanna put the name of the calling algorithm from the user

    start = timeit.default_timer()              #This will return the default time before executing the next line
    function(initial_state)
    stop = timeit.default_timer()               #This will return the default time after executing the above previous line

    export( stop-start)

    print('--------------the Graphical user interface of the "8-Puzzle Game" is working in a new window now ------------------')
    print('-----------------------------------End Of The program-------------------------\n-------------------------------------------------bye..bye!------------------------')
    sol = backtrace()                           # sol=moves
    GUI(initial_state, sol)                     #if the algorithm choosen by the user is A* ,then i will make the Gui for manhatten distance heuristics only , in order not to duplicate the root and destroy the current tk, 34an md5l4 nfsy fe 7warat


function_map ={
    'BFS': bfs,'bfs': bfs,
    'DFS': dfs,'dfs': dfs,
    'A*' : ast, 'a*':ast, 'a star':ast, 'A STAR':ast, 'A Star':ast, 'A star':ast
    }


#--------------------------------inserting the input from the user--------------------------------
algor_type    = input("Enter the algorithm type u wanna use..For ex,type works like these>> BFS,DFS,A* : ")
initial_board = input("Enter your initial state..For ex>> 0,8,7,6,5,4,3,2,1 or 1,2,5,3,4,0,6,7,8 or etc: ")
print('the program is running now! just wait for a second and the program will print your output in details')
main(algor_type,initial_board)
```

2.The "ClassOfStates.py":

```
# Asmaa Gamal
#state_class of Assignment 1
# Electronics & Communications department
#using informed and uninformed search Algorithms to solve 8-puzzle



class State:

    #depth = {} #i won't use it but it is to solve a warning as what stack overflow told me here: 
https://stackoverflow.com/questions/28172008/pycharm-visual-warning-about-unresolved-attribute-reference

    def __init__(self, state, parent, move, depth, cost, key):

        self.state = state

        self.parent = parent

        self.move = move

        self.depth = depth

        self.cost = cost

        self.key = key

        if self.state:
            self.map = ''.join(str(e) for e in self.state)    #map = for loop of the whole elements in all 
2d or 4 d or n dimensions so al map  btmsk each element we te apply 3leh al change elle ana 3awza 23mloh

    def __eq__(self, other): #to transform the num into str to compare and find if equal
        return self.map == other.map

    def __lt__(self, other): #to transform the num into str to compare and find if less than
        return self.map < other.map
```

3.The “GUI.py”:

```
# Asmaa Gamal
#state_class of Assignment 1
# Electronics & Communications department
#using informed and uninformed search Algorithms to solve 8-puzzle

# ---------------------------------------visualization-------------------------------------------
from tkinter import *

tk = Tk()

ANSI_RESET = "\u001B[0m"
ANSI_CYAN = "\u001B[36m"

def print_cyan(msg):
    print(f"{ANSI_CYAN}{msg}{ANSI_RESET}")


def init_button(num):
    if num == 0:
        return Label(tk, text=num, font='Times 25 bold', bg='grey', fg='black', height=4, width=8)
    else:
        return Label(tk, text=num, font='Times 25 bold', bg='cyan3', fg='black', height=4, width=8)


def shuffle(index, target,puzzle):  #puzzle=initial_state
    temp = puzzle[target]
    puzzle[target] = 0
    puzzle[index] = temp

def GUI(puzzle,sol):                #puzzle=initial_state

    tk.title("                  GUI Solver Of The 8-Puzzle Game")
    index = 0

    button0 = init_button(puzzle[0])
    button0.grid(row=3, column=0)
    button1 = init_button(puzzle[1])
    button1.grid(row=3, column=1)
    button2 = init_button(puzzle[2])
    button2.grid(row=3, column=2)
    button3 = init_button(puzzle[3])
    button3.grid(row=4, column=0)
    button4 = init_button(puzzle[4])
    button4.grid(row=4, column=1)
    button5 = init_button(puzzle[5])
    button5.grid(row=4, column=2)
    button6 = init_button(puzzle[6])
    button6.grid(row=5, column=0)
    button7 = init_button(puzzle[7])
    button7.grid(row=5, column=1)
    button8 = init_button(puzzle[8])
    button8.grid(row=5, column=2)


    def fun(index):
        zero_place = puzzle.index(0)
        target = 0
        if zero_place == 0:
            if sol[index] == 'Down':
                target = 3
                button0["text"] = button3["text"]
                button0["bg"] = 'cyan2'
                tk.update()
                button3["text"] = 0
                button3["bg"] = 'grey'
                tk.update()
            elif sol[index] == 'Right':
                target = 1
                button0["text"] = button1["text"]
                button0["bg"] = 'cyan2'
                tk.update()
                button1["text"] = 0
```

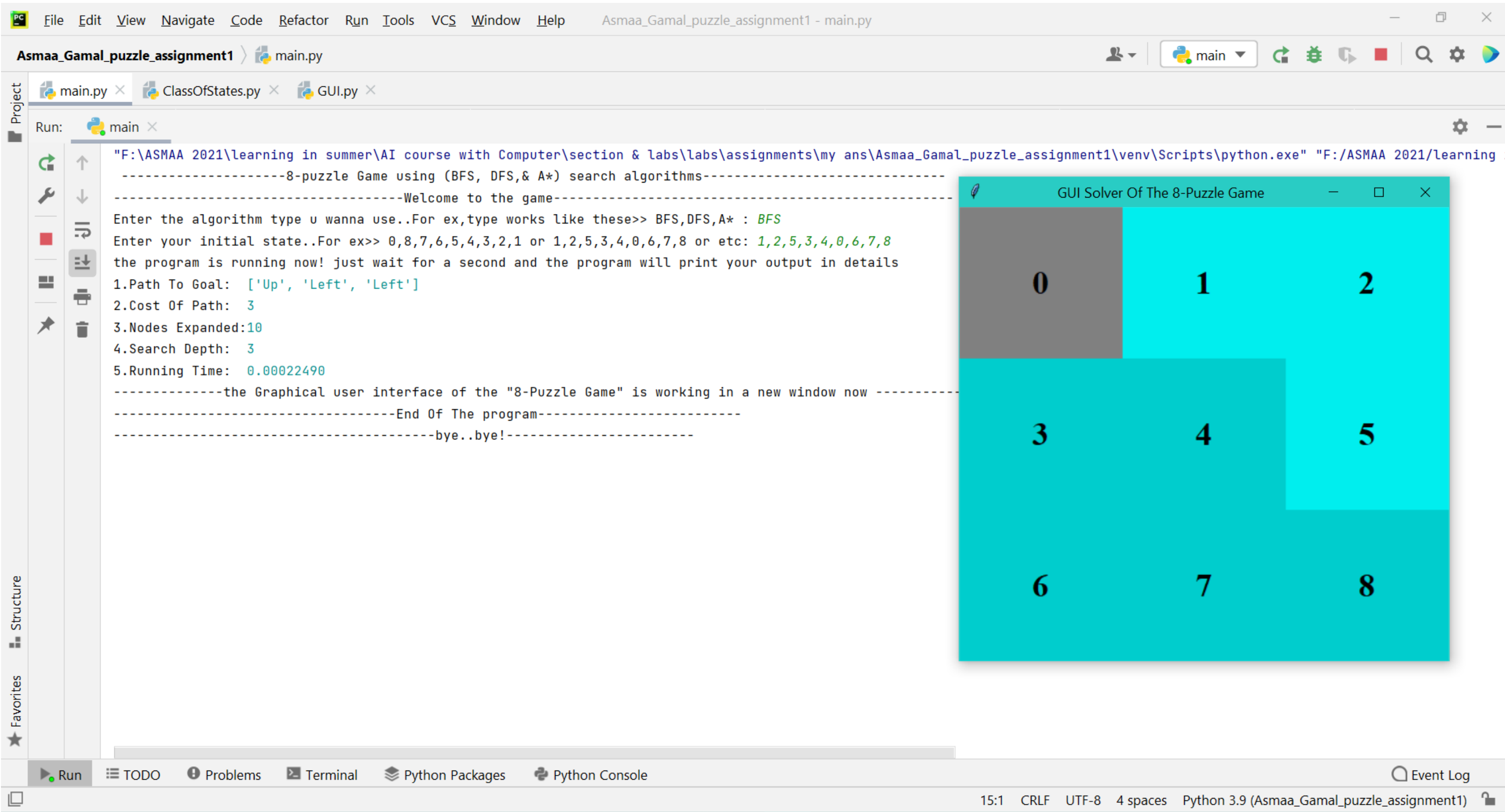
```
            button1["bg"] = 'grey'
            tk.update()
        shuffle(zero_place, target,puzzle)
    elif zero_place == 1:
        if sol[index] == 'Left':
            target = 0
            button1["text"] = button0["text"]
            button1["bg"] = 'cyan2'
            tk.update()
            button0["text"] = 0
            button0["bg"] = 'grey'
            tk.update()
        elif sol[index] == 'Down':
            target = 4
            button1["text"] = button4["text"]
            button1["bg"] = 'cyan2'
            tk.update()
            button4["text"] = 0
            button4["bg"] = 'grey'
            tk.update()
        elif sol[index] == 'Right':
            target = 2
            button1["text"] = button2["text"]
            button1["bg"] = 'cyan2'
            tk.update()
            button2["text"] = 0
            button2["bg"] = 'grey'
            tk.update()
        shuffle(zero_place, target,puzzle)
    elif zero_place == 2:
        if sol[index] == 'Left':
            target = 1
            button2["text"] = button1["text"]
            button2["bg"] = 'cyan2'
            tk.update()
            button1["text"] = 0
            button1["bg"] = 'grey'
            tk.update()
        elif sol[index] == 'Down':
            target = 5
            button2["text"] = button5["text"]
            button2["bg"] = 'cyan2'
            tk.update()
            button5["text"] = 0
            button5["bg"] = 'grey'
            tk.update()
        shuffle(zero_place, target,puzzle)
    elif zero_place == 3:
        if sol[index] == 'Up':
            target = 0
            button3["text"] = button0["text"]
            button3["bg"] = 'cyan2'
            tk.update()
            button0["text"] = 0
            button0["bg"] = 'grey'
            tk.update()
        elif sol[index] == 'Down':
            target = 6
            button3["text"] = button6["text"]
            button3["bg"] = 'cyan2'
            tk.update()
            button6["text"] = 0
            button6["bg"] = 'grey'
            tk.update()
        elif sol[index] == 'Right':
            target = 4
            button3["text"] = button4["text"]
            button3["bg"] = 'cyan2'
            tk.update()
            button4["text"] = 0
            button4["bg"] = 'grey'
            tk.update()
        shuffle(zero_place, target,puzzle)
    elif zero_place == 4:
        if sol[index] == 'Up':
```

```
            target = 1
            button4["text"] = button1["text"]
            button4["bg"] = 'cyan2'
            tk.update()
            button1["text"] = 0
            button1["bg"] = 'grey'
            tk.update()
        elif sol[index] == 'Left':
            target = 3
            button4["text"] = button3["text"]
            button4["bg"] = 'cyan2'
            tk.update()
            button3["text"] = 0
            button3["bg"] = 'grey'
            tk.update()
        elif sol[index] == 'Down':
            target = 7
            button4["text"] = button7["text"]
            button4["bg"] = 'cyan2'
            tk.update()
            button7["text"] = 0
            button7["bg"] = 'grey'
            tk.update()
        elif sol[index] == 'Right':
            target = 5
            button4["text"] = button5["text"]
            button4["bg"] = 'cyan2'
            tk.update()
            button5["text"] = 0
            button5["bg"] = 'grey'
            tk.update()
        shuffle(zero_place, target,puzzle)
    elif zero_place == 5:
        if sol[index] == 'Up':
            target = 2

            button5["text"] = button2["text"]
            button5["bg"] = 'cyan2'
            tk.update()
            button2["text"] = 0
            button2["bg"] = 'grey'
            tk.update()
        elif sol[index] == 'Left':
            target = 4
            button5["text"] = button4["text"]
            button5["bg"] = 'cyan2'
            tk.update()
            button4["text"] = 0
            button4["bg"] = 'grey'
            tk.update()
        elif sol[index] == 'Down':
            target = 8
            button5["text"] = button8["text"]
            button5["bg"] = 'cyan2'
            tk.update()
            button8["text"] = 0
            button8["bg"] = 'grey'
            tk.update()
        shuffle(zero_place, target,puzzle)
    elif zero_place == 6:
        if sol[index] == 'Up':
            target = 3
            button6["text"] = button3["text"]
            button6["bg"] = 'cyan2'
            tk.update()
            button3["text"] = 0
            button3["bg"] = 'grey'
            tk.update()
        elif sol[index] == 'Right':
            target = 7
            button6["text"] = button7["text"]
            button6["bg"] = 'cyan2'
            tk.update()
            button7["text"] = 0
            button7["bg"] = 'grey'
```

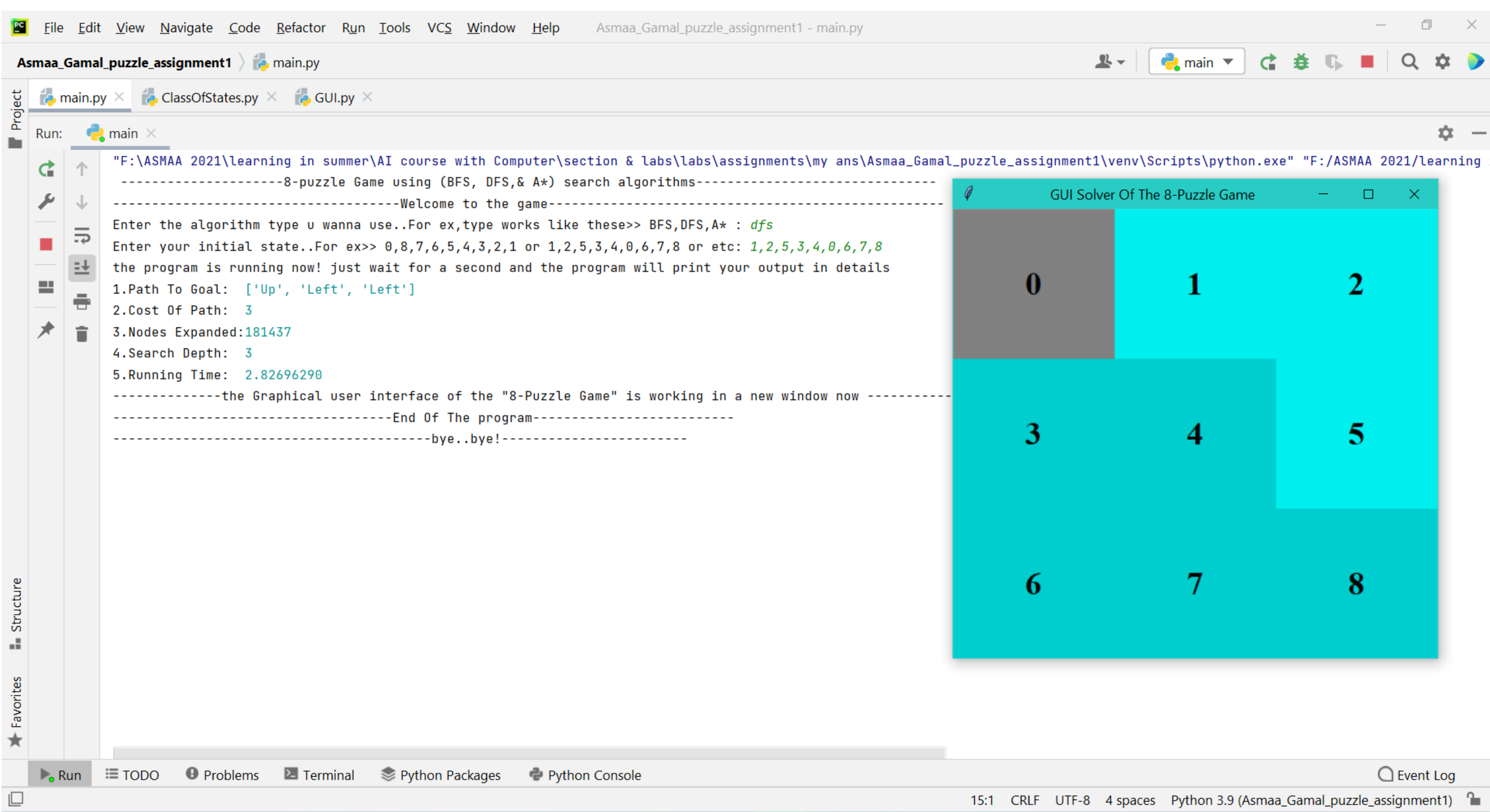
```
            tk.update()
        shuffle(zero_place, target,puzzle)
    elif zero_place == 7:
        if sol[index] == 'Up':
            target = 4
            button7["text"] = button4["text"]
            button7["bg"] = 'cyan2'
            tk.update()
            button4["text"] = 0
            button4["bg"] = 'grey'
            tk.update()
        elif sol[index] == 'Left':
            target = 6
            button7["text"] = button6["text"]
            button7["bg"] = 'cyan2'
            tk.update()
            button6["text"] = 0
            button6["bg"] = 'grey'
            tk.update()
        elif sol[index] == 'Right':
            target = 8
            button7["text"] = button8["text"]
            button7["bg"] = 'cyan2'
            tk.update()
            button8["text"] = 0
            button8["bg"] = 'grey'
            tk.update()
        shuffle(zero_place, target,puzzle)
    elif zero_place == 8:
        if sol[index] == 'Up':
            target = 5
            button8["text"] = button5["text"]
            button8["bg"] = 'cyan2'
            tk.update()
            button5["text"] = 0
            button5["bg"] = 'grey'
            tk.update()
        elif sol[index] == 'Left':
            target = 7
            button8["text"] = button7["text"]
            button8["bg"] = 'cyan2'
            tk.update()
            button7["text"] = 0
            button7["bg"] = 'grey'
            tk.update()
        shuffle(zero_place, target, puzzle)
    if index != len(sol) - 1:
        index += 1
        tk.after(650, fun,index)
    index += 1

tk.after(3000, fun,index)
tk.mainloop()
```

Screenshots Of Some Runs

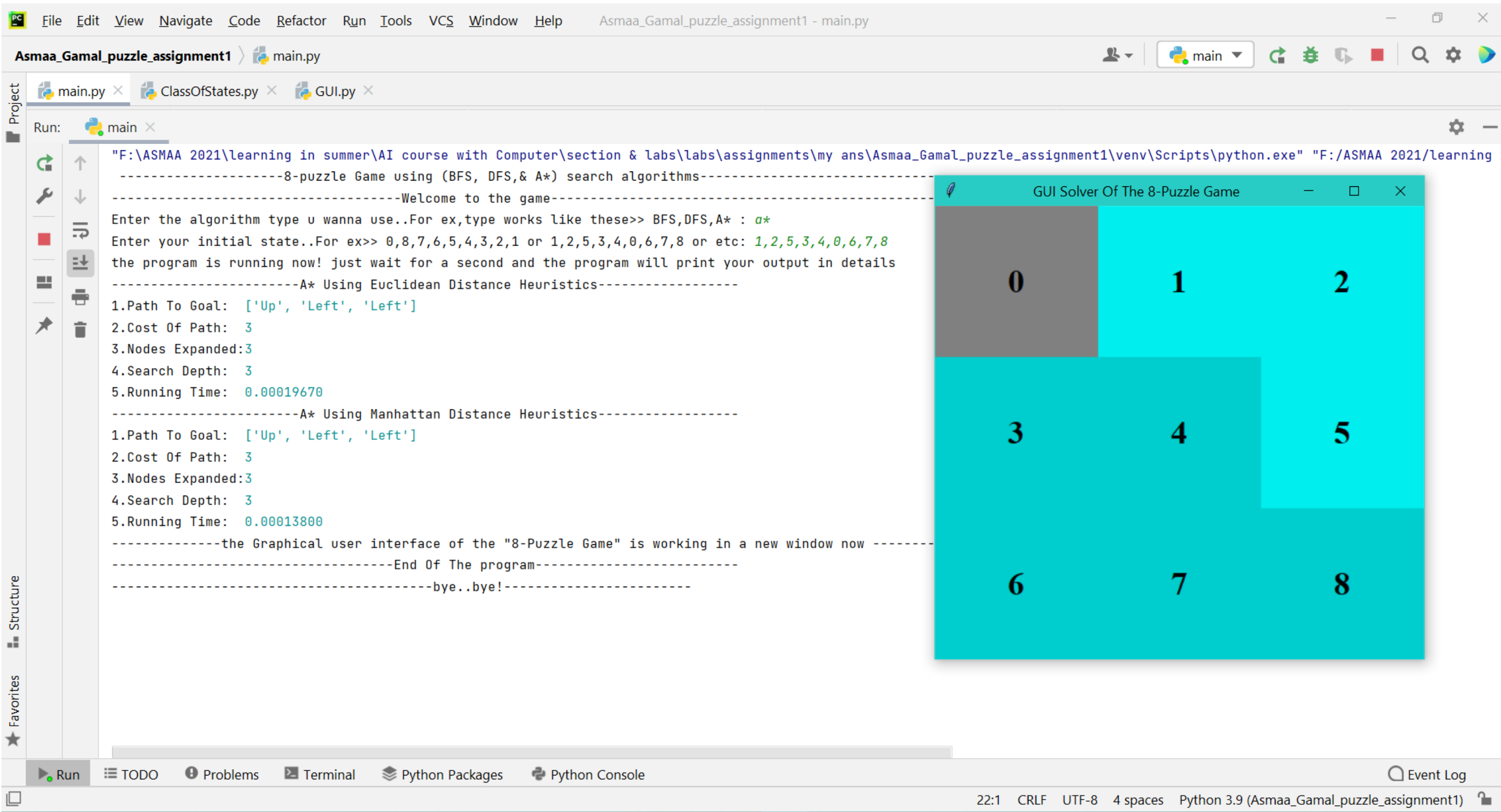
1. BFS



2. DFS



3. A*



Data Structures Used

- Stack (list)
- Queue(list)
- Heap(list)
- Tuples
- Set
- Dictionary

Algorithms

BFS search

```
function BREADTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Queue.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.dequeue()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.enqueue(neighbor)

    return FAILURE
```

DFS search

```
function DEPTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Stack.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.pop()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.push(neighbor)

    return FAILURE
```

A* search

```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Example Shows A summary of my work

[1,2,5,3,4,0,6,7,8]

Data	BFS	DFS	A* (Euclidean)	A* (Manhattan)
Path to goal	['Up', 'Left', 'Left']	['Up', 'Left', 'Left']	['Up', 'Left', 'Left',]	['Up', 'Left', 'Left']
cost	3	3	3	9
Node expanded	10	181437	3	3
Search depth	3	3	3	3
Running time	0.00022490	2.82696290	0.00019670	0.0001380

From this above example, also from more complicated sequence examples outputs in the coming few photos, like: 0,8,7,6,5,4,3,2,1, and finally from the below two equations we can find out that the Manhattan distance heuristics are more admissible than the Euclidean distance heuristics.

1. Manhattan Distance

   It is the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively,

   $$h = abs(current\_cell{:}x - goal{:}x) + abs(current\_cell{:}y - goal{:}y)$$

2. Euclidean Distance

   It is the distance between the current cell and the goal cell using the distance formula

   $$h = sqrt((current\_cell{:}x - goal{:}x)^2 + (current\_cell.y - goal{:}y)^2)$$

```
"F:\ASMAA 2021\learning in summer\AI course with Computer\section & labs\labs\assignments\my ans\Asmaa_Gamal_puzzle_assignment1\venv\Scripts\python.exe" "F:/ASMAA 2021/learning
 ---------------------8-puzzle Game using (BFS, DFS,& A*) search algorithms------------------------------
------------------------------------Welcome to the game-------------------------------------------------
Enter the algorithm type u wanna use..For ex,type works like these>> BFS,DFS,A* : A*
Enter your initial state..For ex>> 0,8,7,6,5,4,3,2,1 or 1,2,5,3,4,0,6,7,8 or etc: 0,8,7,6,5,4,3,2,1
the program is running now! just wait for a second and the program will print your output in details
------------------------A* Using Euclidean Distance Heuristics-----------------
1.Path To Goal:  ['Right', 'Down', 'Down', 'Right', 'Up', 'Up', 'Left', 'Down', 'Down', 'Left', 'Up', 'Right', 'Down', 'Right', 'Up', 'Up', 'Left', 'Left', 'Down', 'Right', 'Up
2.Cost Of Path:  30
3.Nodes Expanded:29367
4.Search Depth:  30
5.Running Time:  1.57665620
------------------------A* Using Manhattan Distance Heuristics-----------------
1.Path To Goal:  ['Right', 'Down', 'Down', 'Right', 'Up', 'Up', 'Left', 'Down', 'Down', 'Left', 'Up', 'Right', 'Down', 'Right', 'Up', 'Up', 'Left', 'Left', 'Down', 'Right', 'Up
2.Cost Of Path:  30
3.Nodes Expanded:12893
4.Search Depth:  30
5.Running Time:  0.90166270
--------------the Graphical user interface of the "8-Puzzle Game" is working in a new window now ------------------
-----------------------------------End Of The program-------------------------
---------------------------------------bye..bye!-----------------------
```

Because the Manhattan output is better in the:

1. Time: its speed or running time is better (as it doesn't need to use the math.sqrt)

2.Memory: Num of "nodes expanded" is better.

Although, the distance of Euclidean diagonal is less than the distance of the absolute differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively

But Manhattan here is more admissible because the only allowed moves here in the 8-puzzle game are the horizontal and the vertical moves only. This means that we can't move diagonally as what the Euclidean distance assumption says. So, the output numbers using the Manhattan distance of this 8-puzzle game here are more admissible^^.