Project 1- Planning for Autonomous Robots  
8-Puzzle solver using Breadth First Search Algorithm

The code for solving the 8-Puzzle by sliding the tile labelled “0” and arranging the digits in ascending order, while putting the 0 in the end is written in Python. The name of the file is **8\_Puzzle\_solver.py**

The goal node has the following shape:

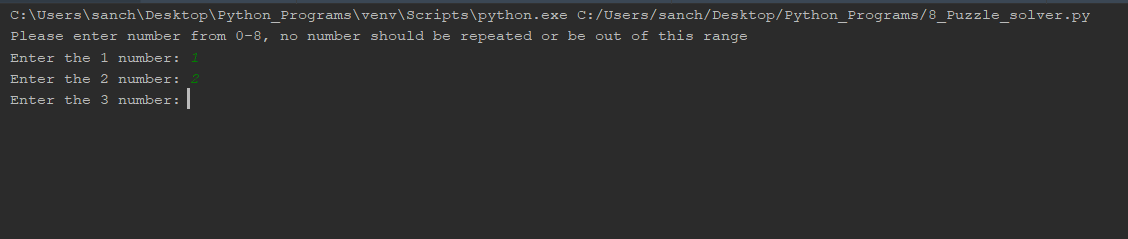
1 2 3

4 5 6

7 8 0

Before running the program, please ensure that Numpy and the OS module is installed. Numpy allows you to create and modify array in python while OS module helps you perform operating system dependent functionality within Python like creating a file, deleting an already made file using its path.

After these packages have been installed, you can go ahead and run the program. Once running, the program will ask the user to input the problem puzzle, the code should look like this:



1 2 3

When it says enter the 2 number, it means this number in the puzzle structure, similar is true for all the other digits

4 5 6

7 8 9

Once all the numbers have been put and they are all within the range of [0-8], the program will start running. If you enter any number which is not in this range, the program will quit with an error message and you will have to re-start putting the number.

The program will then check if any number, which is given to the puzzle is repeated or not, if a number has been repeated, even then an error message will be printed, and the program will quit out as this type of input is not allowed. This is done using the *“def check\_correct\_input(l)”* function

If the input is acceptable, the program will then check if the input puzzle is solvable or not. In the 8-puzzle game there are certain input cases where the final goal node is never reached, when you give such an input, the program will detect it and print a warning message that the following set might not have a solution, but it will still run and check all the nodes. In this condition, all the nodes are explored in approximately 45 min and “Goal Node could not be reached” error is printed out. To check if the input state has a solution or not, the program has the function called *“def check\_solvable(g)”,* in this function, what we do is convert the input state into a single row array and then count the number of times the digits occur in reverse order that they should occur in the goal node. if that counter returns an even value then the puzzle is solvable and if it is odd then it isn't.

Once you have an input which is solvable and is correct, the code starts to run, and “exploring Nodes” statement is printed on your console. This means that the code is running and trying to find the configuration which matches the goal node which is already hard-coded.

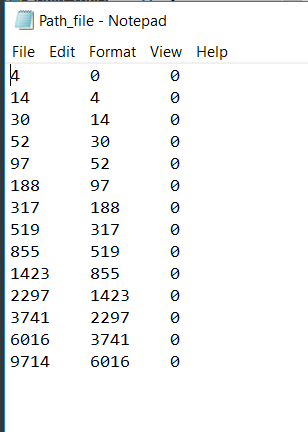
The data structure used to store the puzzle information and the parent information is Node, which has been defined as a class in the python file.

It contains the following attributes:

1. Node Number: each node when it is formed is assigned a unique number
2. Data: the actual puzzle
3. Parent: Parent Node
4. act: the action which was used to create this node
5. Cost: the cost to create this node

As soon as a node is formed, which matches the goal node, the loop breaks, and 4 different functions are called.

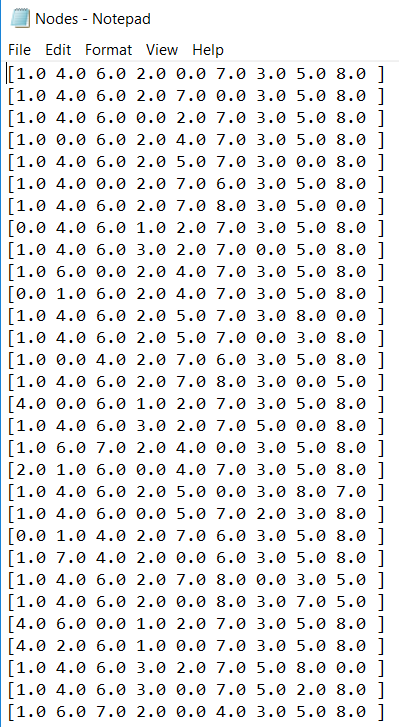
* The first function is used to print the path which should be travelled to reach the goal state from the input puzzle stage, with their corresponding moves.
* The second function is used to print the same path into a text file, this function checks if there is a file name already by the name – “Path\_file.txt”, is there is it deletes it and creates a new Path\_file.txt with new information.



The following is the format in the file:

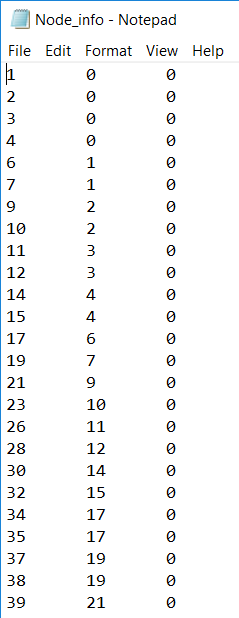
Node Number Parent Number Cost

* The third function is used to print in a text file, the nodes it visited/created while finding the goal node. This file also finds the file named “Nodes.txt”, if it is there it deletes it and then creates it with new information.



Here each row represents a node with its 9 elements

* The fourth function is used to print the node information in a text file, it prints the Node Number, Node Parent number and the cost for each node travelled by the program. It is saved by the name “Nodes\_info.txt”



The following is the format in the file:

Node Number Parent Number Cost

Code is pasted here as well:

*""" 8 Puzzle BFS algorithm  
 Input the unsolved puzzle and the program  
 solves it and creates 3 txt files with solutions"""*import numpy as np # Used to store the digits in an array  
import os # Used to delete the file created by previous running of the program  
  
  
class Node:  
 def \_\_init\_\_(self, node\_no, data, parent, id, cost):  
 self.data = data  
 self.parent = parent  
 self.id = id  
 self.node\_no = node\_no  
 self.cost = cost  
  
  
def get\_initial():  
 print("Please enter number from 0-8, no number should be repeated or be out of this range")  
 initial\_state = np.zeros(9)  
 for i in range(9):  
 states = int(input("Enter the " + str(i + 1) + " number: "))  
 if states < 0 or states > 8:  
 print("Please only enter states which are [0-8], run code again")  
 exit(0)  
 else:  
 initial\_state[i] = np.array(states)  
 return np.reshape(initial\_state, (3, 3))  
  
  
def find\_index(puzzle):  
 i, j = np.where(puzzle == 0)  
 i = int(i)  
 j = int(j)  
 return i, j  
  
  
def move\_left(data):  
 i, j = find\_index(data)  
 if j == 0:  
 return None  
 else:  
 temp\_arr = np.copy(data)  
 temp = temp\_arr[i, j-1]  
 temp\_arr[i, j] = temp  
 temp\_arr[i, j-1] = 0  
 return temp\_arr  
  
  
def move\_right(data):  
 i, j = find\_index(data)  
 if j == 2:  
 return None  
 else:  
 temp\_arr = np.copy(data)  
 temp = temp\_arr[i, j+1]  
 temp\_arr[i, j] = temp  
 temp\_arr[i, j+1] = 0  
 return temp\_arr  
  
  
def move\_up(data):  
 i, j = find\_index(data)  
 if i == 0:  
 return None  
 else:  
 temp\_arr = np.copy(data)  
 temp = temp\_arr[i-1, j]  
 temp\_arr[i, j] = temp  
 temp\_arr[i-1, j] = 0  
 return temp\_arr  
  
  
def move\_down(data):  
 i, j = find\_index(data)  
 if i == 2:  
 return None  
 else:  
 temp\_arr = np.copy(data)  
 temp = temp\_arr[i+1, j]  
 temp\_arr[i, j] = temp  
 temp\_arr[i+1, j] = 0  
 return temp\_arr  
  
  
def move\_tile(action, data):  
 if action == 'up':  
 return move\_up(data)  
 if action == 'down':  
 return move\_down(data)  
 if action == 'left':  
 return move\_left(data)  
 if action == 'right':  
 return move\_right(data)  
 else:  
 return None  
  
  
def print\_states(list\_final): # To print the final states on the console  
 print("printing final solution")  
 for l in list\_final:  
 print("Move : " + str(l.id) + "\n" + "Result : " + "\n" + str(l.data) + "\t" + "node number:" + str(l.node\_no))  
  
  
def write\_path(path\_formed): # To write the final path in the text file  
 if os.path.exists("Path\_file.txt"):  
 os.remove("Path\_file.txt")  
   
 f = open("Path\_file.txt", "a")  
 for node in path\_formed:  
 if node.parent is not None:  
 f.write(str(node.node\_no) + "\t" + str(node.parent.node\_no) + "\t" + str(node.cost)+"\n")  
 f.close()   
  
  
def write\_node\_explored(explored): # To write all the nodes explored by the program  
 if os.path.exists("Nodes.txt"):  
 os.remove("Nodes.txt")  
   
 f = open("Nodes.txt", "a")  
 for element in explored:  
 for i in range(len(element)):   
 f.write(str(element[i])+" ")  
 f.write("\n")  
 f.close()  
  
  
def write\_node\_info(visited): # To write all the info about the nodes explored by the program  
 if os.path.exists("Node\_info.txt"):  
 os.remove("Node\_info.txt")  
   
 f = open("Node\_info.txt", "a")  
 for n in visited:  
 if n.parent is not None:  
 f.write(str(n.node\_no) + "\t" + str(n.parent.node\_no) + "\t" + str(n.cost)+"\n")  
 f.close()  
  
  
def path(node): # To find the path from the goal node to the starting node  
 p = [] # Empty list  
 p.append(node)  
 parent\_node = node.parent  
 while parent\_node is not None:  
 p.append(parent\_node)  
 parent\_node = parent\_node.parent  
 return list(reversed(p))  
  
  
def exploring\_nodes(node):  
 print("Exploring Nodes")  
 actions = ["down", "up", "left", "right"]  
 goal\_node = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])  
 node\_q = [node]  
 final\_nodes = []  
 visited = []  
 final\_nodes.append(node\_q[0].data.tolist()) # Only writing data of nodes in seen  
 node\_counter = 0 # To define a unique ID to all the nodes formed  
  
 while node\_q:  
 current\_root = node\_q.pop(0) # Pop the element 0 from the list  
 if current\_root.data.tolist() == goal\_node.tolist():  
 print("Goal reached")  
 return current\_root, final\_nodes, visited  
   
 for move in actions:  
 temp\_data = move\_tile(move, current\_root.data)  
 if temp\_data is not None:  
 node\_counter += 1  
 child\_node = Node(node\_counter, np.array(temp\_data), current\_root, move, 0) # Create a child node  
   
 if child\_node.data.tolist() not in final\_nodes: # Add the child node data in final node list  
 node\_q.append(child\_node)  
 final\_nodes.append(child\_node.data.tolist())  
 visited.append(child\_node)  
 if child\_node.data.tolist() == goal\_node.tolist():  
 print("Goal\_reached")  
 return child\_node, final\_nodes, visited  
 return None, None, None # return statement if the goal node is not reached  
  
  
def check\_correct\_input(l):  
 array = np.reshape(l, 9)  
 for i in range(9):  
 counter\_appear = 0  
 f = array[i]  
 for j in range(9):  
 if f == array[j]:  
 counter\_appear += 1  
 if counter\_appear >= 2:  
 print("invalid input, same number entered 2 times")  
 exit(0)  
  
  
def check\_solvable(g):  
 arr = np.reshape(g, 9)  
 counter\_states = 0  
 for i in range(9):  
 if not arr[i] == 0:  
 check\_elem = arr[i]  
 for x in range(i+1, 9):  
 if check\_elem < arr[x] or arr[x] == 0:  
 continue  
 else:  
 counter\_states += 1  
 if counter\_states % 2 == 0:  
 print("The puzzle is solvable, generating path")  
 else:  
 print("The puzzle is insolvable, still creating nodes")  
  
  
# Final Running of the Code  
  
# uncomment the line below to run it for a fixed data input and comment the line below it  
#k = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])  
k = get\_initial()  
  
check\_correct\_input(k)  
check\_solvable(k)  
  
root = Node(0, k, None, None, 0)  
  
# BFS implementation call  
goal, s, v = exploring\_nodes(root)  
  
if goal is None and s is None and v is None:  
 print("Goal State could not be reached, Sorry")  
else:  
 # Print and write the final output  
 print\_states(path(goal))  
 write\_path(path(goal))  
 write\_node\_explored(s)  
 write\_node\_info(v)