**Artificial Intelligence – Experiments**

**Exp no 1: Implementation of toy problems(Tic Tac Toe)**

**Link:**

<https://medium.com/byte-tales/the-classic-tic-tac-toe-game-in-python-3-1427c68b8874>

**Source Code:**

#Implementation of Two Player Tic-Tac-Toe game in Python.

''' We will make the board using dictionary

in which keys will be the location(i.e : top-left,mid-right,etc.)

and initialliy it's values will be empty space and then after every move

we will change the value according to player's choice of move. '''

theBoard = {'7': ' ' , '8': ' ' , '9': ' ' ,

'4': ' ' , '5': ' ' , '6': ' ' ,

'1': ' ' , '2': ' ' , '3': ' ' }

board\_keys = []

for key in theBoard:

board\_keys.append(key)

''' We will have to print the updated board after every move in the game and

thus we will make a function in which we'll define the printBoard function

so that we can easily print the board everytime by calling this function. '''

def printBoard(board):

print(board['7'] + '|' + board['8'] + '|' + board['9'])

print('-+-+-')

print(board['4'] + '|' + board['5'] + '|' + board['6'])

print('-+-+-')

print(board['1'] + '|' + board['2'] + '|' + board['3'])

# Now we'll write the main function which has all the gameplay functionality.

def game():

turn = 'X'

count = 0

for i in range(10):

printBoard(theBoard)

print("It's your turn," + turn + ".Move to which place?")

move = input()

if theBoard[move] == ' ':

theBoard[move] = turn

count += 1

else:

print("That place is already filled.\nMove to which place?")

continue

# Now we will check if player X or O has won,for every move after 5 moves.

if count >= 5:

if theBoard['7'] == theBoard['8'] == theBoard['9'] != ' ': # across the top

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['4'] == theBoard['5'] == theBoard['6'] != ' ': # across the middle

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['1'] == theBoard['2'] == theBoard['3'] != ' ': # across the bottom

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['1'] == theBoard['4'] == theBoard['7'] != ' ': # down the left side

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['2'] == theBoard['5'] == theBoard['8'] != ' ': # down the middle

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['3'] == theBoard['6'] == theBoard['9'] != ' ': # down the right side

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['7'] == theBoard['5'] == theBoard['3'] != ' ': # diagonal

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

elif theBoard['1'] == theBoard['5'] == theBoard['9'] != ' ': # diagonal

printBoard(theBoard)

print("\nGame Over.\n")

print(" \*\*\*\* " +turn + " won. \*\*\*\*")

break

# If neither X nor O wins and the board is full, we'll declare the result as 'tie'.

if count == 9:

print("\nGame Over.\n")

print("It's a Tie!!")

# Now we have to change the player after every move.

if turn =='X':

turn = 'O'

else:

turn = 'X'

# Now we will ask if player wants to restart the game or not.

restart = input("Do want to play Again?(y/n)")

if restart == "y" or restart == "Y":

for key in board\_keys:

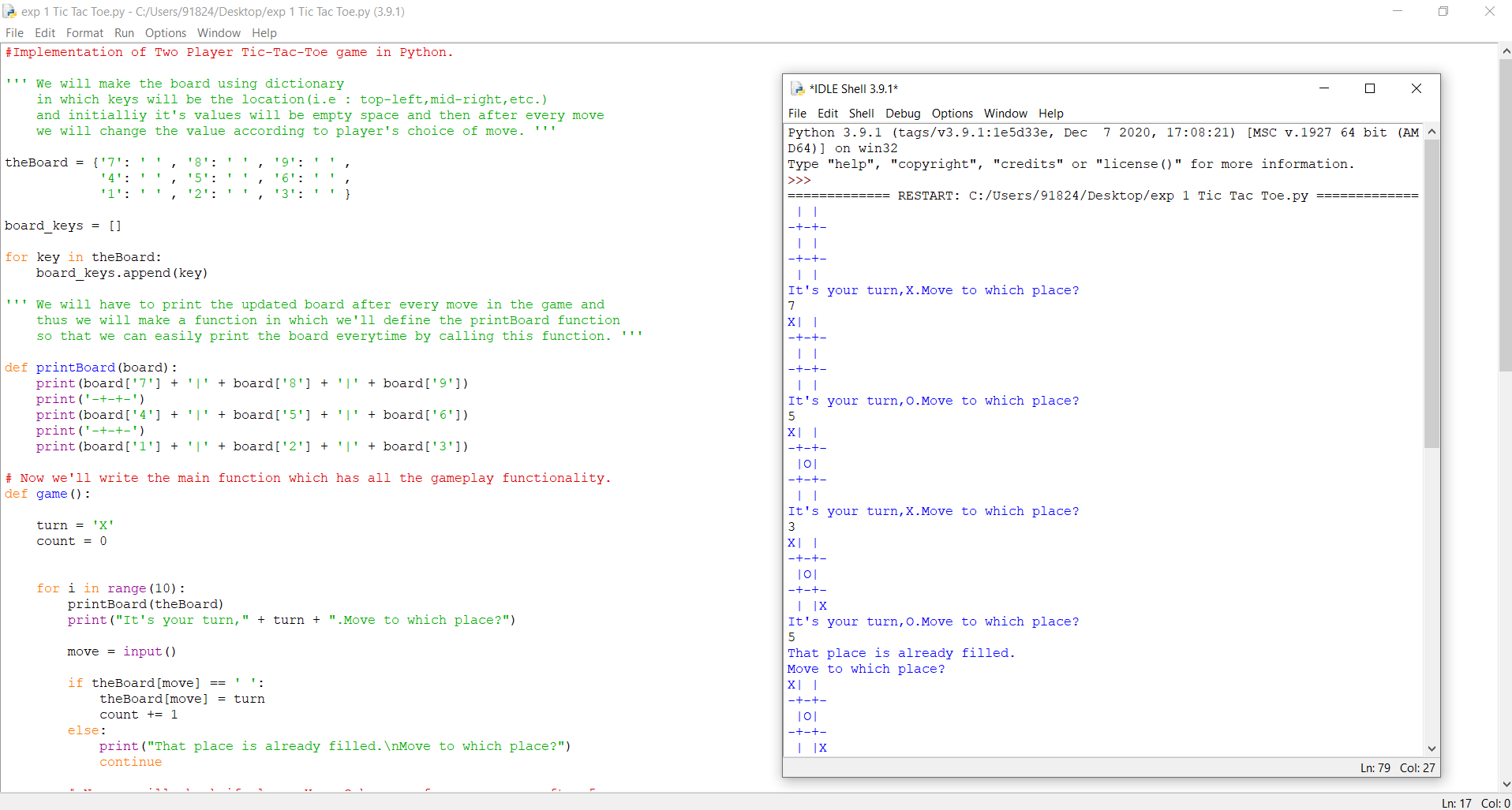
theBoard[key] = " "

game()

if \_\_name\_\_ == "\_\_main\_\_":

game()

**Output:**



**Exp no 2: Developing agent programs for real world problems (8-puzzle)**

**Link:**

<https://gist.github.com/flatline/838202>

**Source Code:**

# Solves a randomized 8-puzzle using A\* algorithm with plug-in heuristics

import random

import math

\_goal\_state = [[1,2,3],

[4,5,6],

[7,8,0]]

def index(item, seq):

"""Helper function that returns -1 for non-found index value of a seq"""

if item in seq:

return seq.index(item)

else:

return -1

class EightPuzzle:

def \_\_init\_\_(self):

# heuristic value

self.\_hval = 0

# search depth of current instance

self.\_depth = 0

# parent node in search path

self.\_parent = None

self.adj\_matrix = []

for i in range(3):

self.adj\_matrix.append(\_goal\_state[i][:])

def \_\_eq\_\_(self, other):

if self.\_\_class\_\_ != other.\_\_class\_\_:

return False

else:

return self.adj\_matrix == other.adj\_matrix

def \_\_str\_\_(self):

res = ''

for row in range(3):

res += ' '.join(map(str, self.adj\_matrix[row]))

res += '\r\n'

return res

def \_clone(self):

p = EightPuzzle()

for i in range(3):

p.adj\_matrix[i] = self.adj\_matrix[i][:]

return p

def \_get\_legal\_moves(self):

"""Returns list of tuples with which the free space may

be swapped"""

# get row and column of the empty piece

row, col = self.find(0)

free = []

# find which pieces can move there

if row > 0:

free.append((row - 1, col))

if col > 0:

free.append((row, col - 1))

if row < 2:

free.append((row + 1, col))

if col < 2:

free.append((row, col + 1))

return free

def \_generate\_moves(self):

free = self.\_get\_legal\_moves()

zero = self.find(0)

def swap\_and\_clone(a, b):

p = self.\_clone()

p.swap(a,b)

p.\_depth = self.\_depth + 1

p.\_parent = self

return p

return map(lambda pair: swap\_and\_clone(zero, pair), free)

def \_generate\_solution\_path(self, path):

if self.\_parent == None:

return path

else:

path.append(self)

return self.\_parent.\_generate\_solution\_path(path)

def solve(self, h):

"""Performs A\* search for goal state.

h(puzzle) - heuristic function, returns an integer

"""

def is\_solved(puzzle):

return puzzle.adj\_matrix == \_goal\_state

openl = [self]

closedl = []

move\_count = 0

while len(openl) > 0:

x = openl.pop(0)

move\_count += 1

if (is\_solved(x)):

if len(closedl) > 0:

return x.\_generate\_solution\_path([]), move\_count

else:

return [x]

succ = x.\_generate\_moves()

idx\_open = idx\_closed = -1

for move in succ:

# have we already seen this node?

idx\_open = index(move, openl)

idx\_closed = index(move, closedl)

hval = h(move)

fval = hval + move.\_depth

if idx\_closed == -1 and idx\_open == -1:

move.\_hval = hval

openl.append(move)

elif idx\_open > -1:

copy = openl[idx\_open]

if fval < copy.\_hval + copy.\_depth:

# copy move's values over existing

copy.\_hval = hval

copy.\_parent = move.\_parent

copy.\_depth = move.\_depth

elif idx\_closed > -1:

copy = closedl[idx\_closed]

if fval < copy.\_hval + copy.\_depth:

move.\_hval = hval

closedl.remove(copy)

openl.append(move)

closedl.append(x)

openl = sorted(openl, key=lambda p: p.\_hval + p.\_depth)

# if finished state not found, return failure

return [], 0

def shuffle(self, step\_count):

for i in range(step\_count):

row, col = self.find(0)

free = self.\_get\_legal\_moves()

target = random.choice(free)

self.swap((row, col), target)

row, col = target

def find(self, value):

"""returns the row, col coordinates of the specified value

in the graph"""

if value < 0 or value > 8:

raise Exception("value out of range")

for row in range(3):

for col in range(3):

if self.adj\_matrix[row][col] == value:

return row, col

def peek(self, row, col):

"""returns the value at the specified row and column"""

return self.adj\_matrix[row][col]

def poke(self, row, col, value):

"""sets the value at the specified row and column"""

self.adj\_matrix[row][col] = value

def swap(self, pos\_a, pos\_b):

"""swaps values at the specified coordinates"""

temp = self.peek(\*pos\_a)

self.poke(pos\_a[0], pos\_a[1], self.peek(\*pos\_b))

self.poke(pos\_b[0], pos\_b[1], temp)

def heur(puzzle, item\_total\_calc, total\_calc):

"""

Heuristic template that provides the current and target position for each number and the

total function.

Parameters:

puzzle - the puzzle

item\_total\_calc - takes 4 parameters: current row, target row, current col, target col.

Returns int.

total\_calc - takes 1 parameter, the sum of item\_total\_calc over all entries, and returns int.

This is the value of the heuristic function

"""

t = 0

for row in range(3):

for col in range(3):

val = puzzle.peek(row, col) - 1

target\_col = val % 3

target\_row = val / 3

# account for 0 as blank

if target\_row < 0:

target\_row = 2

t += item\_total\_calc(row, target\_row, col, target\_col)

return total\_calc(t)

#some heuristic functions, the best being the standard manhattan distance in this case, as it comes

#closest to maximizing the estimated distance while still being admissible.

def h\_manhattan(puzzle):

return heur(puzzle,

lambda r, tr, c, tc: abs(tr - r) + abs(tc - c),

lambda t : t)

def h\_manhattan\_lsq(puzzle):

return heur(puzzle,

lambda r, tr, c, tc: (abs(tr - r) + abs(tc - c))\*\*2,

lambda t: math.sqrt(t))

def h\_linear(puzzle):

return heur(puzzle,

lambda r, tr, c, tc: math.sqrt(math.sqrt((tr - r)\*\*2 + (tc - c)\*\*2)),

lambda t: t)

def h\_linear\_lsq(puzzle):

return heur(puzzle,

lambda r, tr, c, tc: (tr - r)\*\*2 + (tc - c)\*\*2,

lambda t: math.sqrt(t))

def h\_default(puzzle):

return 0

def main():

p = EightPuzzle()

p.shuffle(20)

print p

path, count = p.solve(h\_manhattan)

path.reverse()

for i in path:

print i

print "Solved with Manhattan distance exploring", count, "states"

path, count = p.solve(h\_manhattan\_lsq)

print "Solved with Manhattan least squares exploring", count, "states"

path, count = p.solve(h\_linear)

print "Solved with linear distance exploring", count, "states"

path, count = p.solve(h\_linear\_lsq)

print "Solved with linear least squares exploring", count, "states"

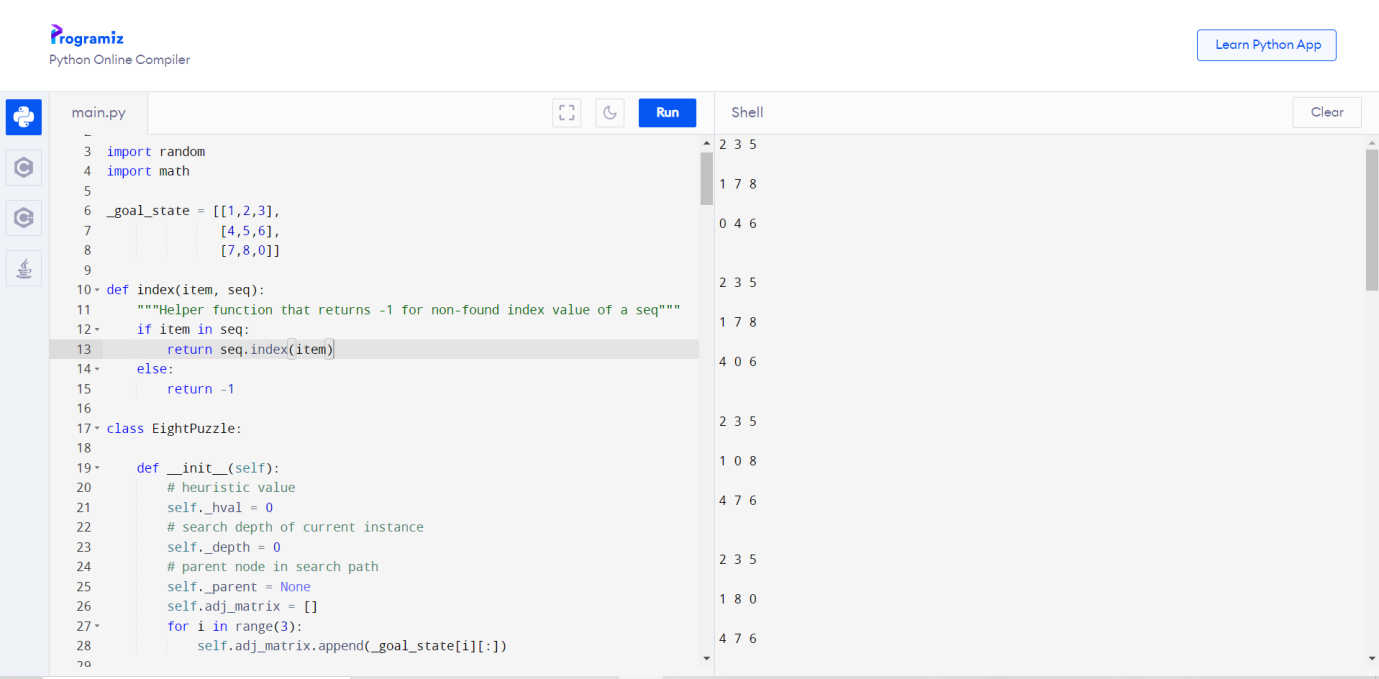
# path, count = p.solve(heur\_default)

# print "Solved with BFS-equivalent in", count, "moves"

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Output:**



# Exp no 3: Implementation of constraint satisfaction problems(Cryptarithmetic Problem)

# Link:

# <https://www.tutorialandexample.com/cryptarithmetic-problem/>

# Source Code:

// CPP program for solving cryptographic puzzles

#include <bits/stdc++.h>

using namespace std;

// vector stores 1 corresponding to index

// number which is already assigned

// to any char, otherwise stores 0

vector<int> use(10);

// structure to store char and its corresponding integer

struct node

{

char c;

int v;

};

// function check for correct solution

int check(node\* nodeArr, const int count, string s1,

string s2, string s3)

{

int val1 = 0, val2 = 0, val3 = 0, m = 1, j, i;

// calculate number corresponding to first string

for (i = s1.length() - 1; i >= 0; i--)

{

char ch = s1[i];

for (j = 0; j < count; j++)

if (nodeArr[j].c == ch)

break;

val1 += m \* nodeArr[j].v;

m \*= 10;

}

m = 1;

// calculate number corresponding to second string

for (i = s2.length() - 1; i >= 0; i--)

{

char ch = s2[i];

for (j = 0; j < count; j++)

if (nodeArr[j].c == ch)

break;

val2 += m \* nodeArr[j].v;

m \*= 10;

}

m = 1;

// calculate number corresponding to third string

for (i = s3.length() - 1; i >= 0; i--)

{

char ch = s3[i];

for (j = 0; j < count; j++)

if (nodeArr[j].c == ch)

break;

val3 += m \* nodeArr[j].v;

m \*= 10;

}

// sum of first two number equal to third return true

if (val3 == (val1 + val2))

return 1;

// else return false

return 0;

}

// Recursive function to check solution for all permutations

bool permutation(const int count, node\* nodeArr, int n,

string s1, string s2, string s3)

{

// Base case

if (n == count - 1)

{

// check for all numbers not used yet

for (int i = 0; i < 10; i++)

{

// if not used

if (use[i] == 0)

{

// assign char at index n integer i

nodeArr[n].v = i;

// if solution found

if (check(nodeArr, count, s1, s2, s3) == 1)

{

cout << "\nSolution found: ";

for (int j = 0; j < count; j++)

cout << " " << nodeArr[j].c << " = "

<< nodeArr[j].v;

return true;

}

}

}

return false;

}

for (int i = 0; i < 10; i++)

{

// if ith integer not used yet

if (use[i] == 0)

{

// assign char at index n integer i

nodeArr[n].v = i;

// mark it as not available for other char

use[i] = 1;

// call recursive function

if (permutation(count, nodeArr, n + 1, s1, s2, s3))

return true;

// backtrack for all other possible solutions

use[i] = 0;

}

}

return false;

}

bool solveCryptographic(string s1, string s2,

string s3)

{

// count to store number of unique char

int count = 0;

// Length of all three strings

int l1 = s1.length();

int l2 = s2.length();

int l3 = s3.length();

// vector to store frequency of each char

vector<int> freq(26);

for (int i = 0; i < l1; i++)

++freq[s1[i] - 'A'];

for (int i = 0; i < l2; i++)

++freq[s2[i] - 'A'];

for (int i = 0; i < l3; i++)

++freq[s3[i] - 'A'];

// count number of unique char

for (int i = 0; i < 26; i++)

if (freq[i] > 0)

count++;

// solution not possible for count greater than 10

if (count > 10)

{

cout << "Invalid strings";

return 0;

}

// array of nodes

node nodeArr[count];

// store all unique char in nodeArr

for (int i = 0, j = 0; i < 26; i++)

{

if (freq[i] > 0)

{

nodeArr[j].c = char(i + 'A');

j++;

}

}

return permutation(count, nodeArr, 0, s1, s2, s3);

}

// Driver function

int main()

{

string s1 = "SEND";

string s2 = "MORE";

string s3 = "MONEY";

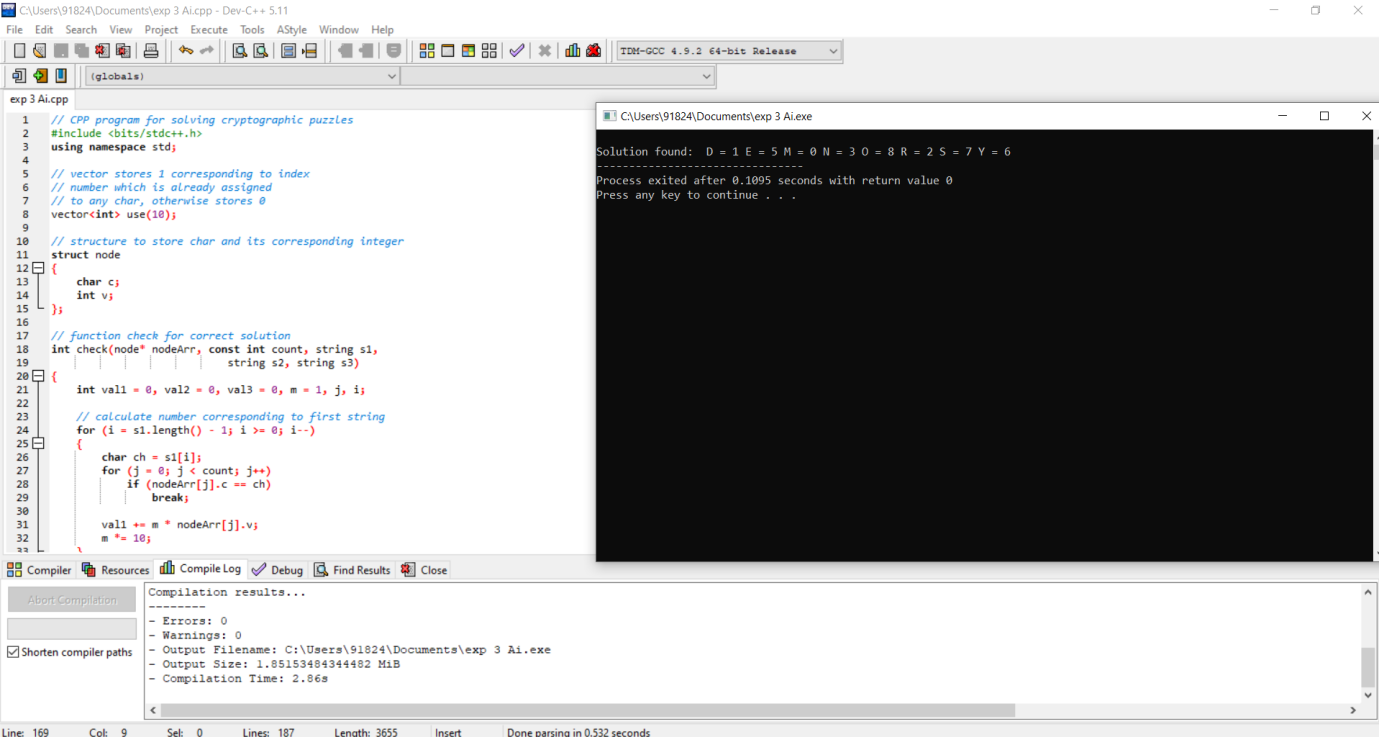
if (solveCryptographic(s1, s2, s3) == false)

cout << "No solution";

return 0;

}

**Output:**



//checking sub matrix

int row\_start = (r/3)\*3;

int col\_start = (c/3)\*3;

for(i=row\_start;i<row\_start+3;i++)

{

for(j=col\_start;j<col\_start+3;j++)

{

if(matrix[i][j]==n)

return 0;

}

}

return 1;

}

//function to solve sudoku

//using backtracking

int solve\_sudoku()

{

int row;

int col;

//if all cells are assigned then the sudoku is already solved

//pass by reference because number\_unassigned will change the values of row and col

if(number\_unassigned(&row, &col) == 0)

return 1;

int n,i;

//number between 1 to 9

for(i=1;i<=SIZE;i++)

{

//if we can assign i to the cell or not

//the cell is matrix[row][col]

if(is\_safe(i, row, col))

{

matrix[row][col] = i;

//backtracking

if(solve\_sudoku())

return 1;

//if we can't proceed with this solution

//reassign the cell

matrix[row][col]=0;

}

}

return 0;

}

int main()

{

if (solve\_sudoku())

print\_sudoku();

else

printf("No solution\n");

return 0;

}

**Output:**

