# AI(2180703)

## Tutorial - 2

Name : harshil khandhar
Enrollment No. : 170200107053
Division/Batch : E/E4

Q. Write a program to implement BFS (for 8 puzzle problem or Water Jug problem or any AI search problem).

**Code (BFS.py):**

```
#Write a program to implement BFS (for 8 puzzle
#problem or Water Jug problem or any AI search
#problem)
import sys
import numpy as np
from collections import deque
import heapq


class Board:
	parent = None
	state = None
	operator = None
	depth = 0
	zero = None
	cost = 0

	def __init__(self, state, parent = None, operator = None, depth = 0):
		self.parent = parent
		self.state = np.array(state)
		self.operator =  operator
		self.depth = depth
		self.zero = self.find_0()
		self.cost = self.depth + self.manhattan()

	def __lt__(self, other):
		if self.cost != other.cost:
			return self.cost < other.cost
		else:
			op_pr = {'Up': 0, 'Down': 1, 'Left': 2, 'Right': 3}
			return op_pr[self.operator] < op_pr[other.operator]

	def __str__(self):
		return str(self.state[:3]) + '\n' \
				+ str(self.state[3:6]) + '\n' \
				+ str(self.state[6:]) + ' ' \
				+ str(self.depth) + str(self.operator) + '\n'

	def goal_test(self):
		if np.array_equal(self.state, np.arange(9)):
			return True
		else:
			return False

	def find_0(self):
		for i in range(9):
```

```
            if self.state[i] == 0:
                return i

    def manhattan(self):
        state = self.index(self.state)
        goal = self.index(np.arange(9))
        a = abs(state // 3 - goal // 3)
        b = abs(state % 3 - goal % 3)
        tempAbs = (a + b)[1:]
        return sum(tempAbs)

    def index(self, state):
        index = np.array(range(9))
        for x, y in enumerate(state):
            index[y] = x
        return index

    def swap(self, i, j):
        new_state = np.array(self.state)
        new_state[i], new_state[j] = new_state[j], new_state[i]
        return new_state

    def up(self):
        if self.zero > 2:
            t = self.swap(self.zero,self.zero-3)
            return Board(t, self, 'Up', self.depth + 1)
        else:
            return None

    def down(self):
        if self.zero < 6:
            t = self.swap(self.zero, self.zero + 3)
            return Board(t,self, 'Down', self.depth + 1)
        else:
            return None

    def left(self):
        if self.zero % 3 != 0:
            t = self.swap(self.zero, self.zero - 1)
            return Board(t, self, 'Left', self.depth + 1)
        else:
            return None

    def right(self):
        if (self.zero + 1) % 3 != 0:
            t = self.swap(self.zero, self.zero + 1)
            return Board(t, self, 'Right', self.depth + 1)
        else:
            return None

    def neighbors(self):
        neighbors = []
        neighbors.append(self.up())
        neighbors.append(self.down())
        neighbors.append(self.left())
        neighbors.append(self.right())
        return list(filter(None, neighbors))

    __repr__ = __str__


class Solver:
    soln = None
    path = None
```

```
        nodes_expanded = 0
        max_depth = 0

        def ancestral_chain(self):
                current = self.soln
                chain = [current]
                while current.parent != None:
                        chain.append(current.parent)
                        current = current.parent
                return chain

        def path(self):
                path = [t.operator for t in self.ancestral_chain()[-2::-1]]
                return path

        def bfs(self, state):
                frontier = deque()
                frontier.append(state)
                froxplored = set()
                while frontier:
                        board = frontier.popleft()
                        froxplored.add(tuple(board.state))
                        if board.goal_test():
                                self.soln = board
                                self.path = self.path()
                                a = len(froxplored)
                                b = len(frontier)-1
                                self.nodes_expanded = a - b
                                return self.soln
                        for neighbor in board.neighbors():
                                if tuple(neighbor.state) not in froxplored:
                                        frontier.append(neighbor)
                                        froxplored.add(tuple(neighbor.state))
                                        a = self.max_depth
                                        b = neighbor.depth
                                        self.max_depth = max(a,b)
                return None

def main():

        p = Board(np.array(eval(sys.argv[2])))
        s = Solver()
        soln = s.bfs(p)

        count = 1
        print('\nBelow is The Goal State(0 is blank)')
        print('\n0 1 2\n3 4 5\n6 7 8')
        print('\nFollow the below Direction To Solve the problem\n')

        if not s.path:
                print('No Moves Required.Already in solved state...!!!')
                return

        for data in s.path:
                print(str(count)+'.'+data)
                count += 1

if __name__ == "__main__":
        main()
```

**Output :**

```
C:\Users\hp\Desktop\SEM-8\AI\PRACTICAL>py BFS.py bfs 8,3,6,1,4,2,0,5,7

Below is The Goal State(0 is blank)

0 1 2
3 4 5
6 7 8

Follow the below Direction To Solve the problem

1.Right
2.Up
3.Up
4.Left
5.Down
6.Right
7.Up
8.Right
9.Down
10.Left
11.Up
12.Left
13.Down
14.Down
15.Right
16.Up
17.Left
18.Down
19.Right
20.Right
21.Up
22.Left
23.Left
24.Up

C:\Users\hp\Desktop\SEM-8\AI\PRACTICAL>
```