```
'''
    Name : Ayush Pandey
    Roll No : 3317

    ASSIGNMENT-2
    Problem Statement :
        Implement A-star algorithm for any game search problem.
'''

import time
from queue import PriorityQueue, Queue, LifoQueue

class PuzzleSolver:
    def __init__(self, n=3):
        self.boardList = []
        self.n = n
        self.goalState = None

    def solveAStart(self):
        startTime = time.time()
        board = Board(self.boardList, goalState=self.goalState, n=self.n)
        print("Start State .............")
        print(board)

        goal = Board(board.goalState, goalState=None, n=self.n)
        print("Goal State ..............")
        print(goal)

        queue = PriorityQueue()
        queue.put(board.getPriority(0))
        i = 1

        while not queue.empty():
            board = queue.get()[2]
```

```
            if not board.isGoal():
                for neighbour in board.getNeighbours():
                    if neighbour != board.previous:
                        queue.put(neighbour.getPriority(i))
                        i += 1
            else:
                self.analytics("A star", board.move, i, time.time() - startTime,
board)
                return

    def solveBFS(self):
        startTime = time.time()
        board = Board(self.boardList, goalState=self.goalState, n=self.n)

        visited = list()
        queue = Queue()
        queue.put(board.getPriority(0)[2])
        i = 1

        while not queue.empty():
            board = queue.get()
            if not board.isGoal():
                for neighbour in board.getNeighbours():
                    if neighbour not in visited:
                        visited.append(neighbour)
                        queue.put(neighbour)
                        i += 1
            else:
                self.analytics("BFS", board.move, i, time.time() - startTime,
board)
                return

    def solveDFS(self):
        startTime = time.time()
        board = Board(self.boardList, goalState=self.goalState, n=self.n)
```

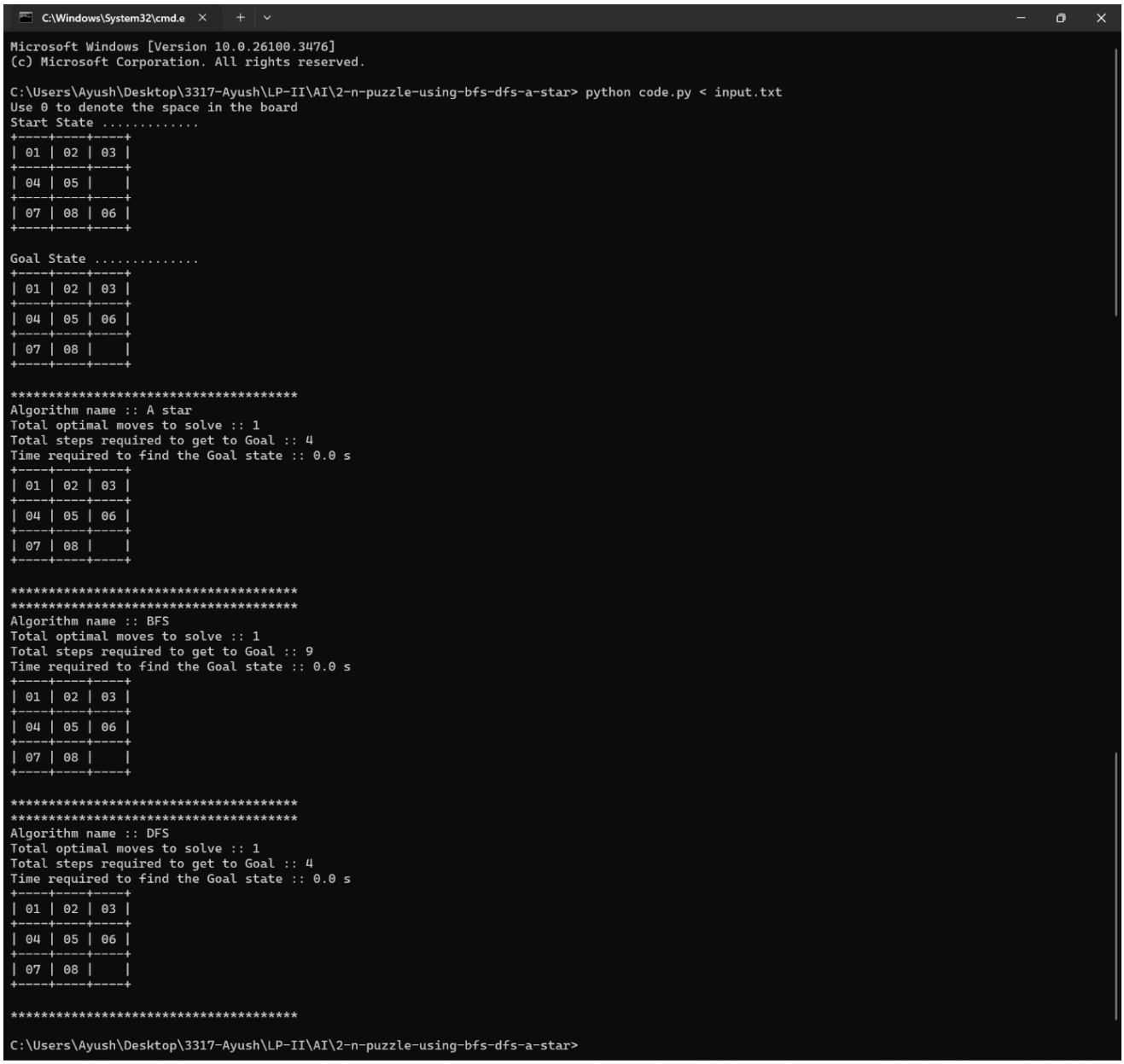
```
        visited = list()
        queue = LifoQueue()
        queue.put(board.getPriority(0)[2])
        i = 1

        while not queue.empty():
            board = queue.get()
            if not board.isGoal():
                for neighbour in board.getNeighbours():
                    if neighbour not in visited:
                        visited.append(neighbour)
                        queue.put(neighbour)
                        i += 1
            else:
                self.analytics("DFS", board.move, i, time.time() - startTime,
board)
                return

    def start(self, goalState=False):
        # print("Enter input board")
        for i in range(0, self.n * self.n):
            self.boardList.append(int(input()))

        if goalState:
            self.goalState = []
            # print("Enter goal board (including space)")
            for i in range(0, self.n * self.n):
                self.goalState.append(int(input()))

        return self

    def analytics(self, method, moves, steps, executionTime, board):
        print("************************************")
```

```
        print(f"Algorithm name :: {method}")
        print(f"Total optimal moves to solve :: {moves}")
        print(f"Total steps required to get to Goal :: {steps}")
        print(f"Time required to find the Goal state :: {round(executionTime, 3)}
s")
        print(board)
        print("************************************")


class Board:
    def __init__(self, board, goalState=None, move=0, previous=None, n=3):
        self.board = board
        self.move = move
        self.previous = previous
        self.n = n
        self.goalState = list()

        if goalState is None:
            for i in range(1, self.n * self.n):
                self.goalState.append(i)
            self.goalState.append(0)
        else:
            self.goalState = goalState

    def __str__(self):
        string = ''
        string = string + ('+----' * self.n) + '+' + '\n'
        for i in range(self.n):
            for j in range(self.n):
                tile = self.board[i * self.n + j]
                string = string + '| {} '.format('  ' if tile == 0 else
str(tile).zfill(2))
            string = string + '|\n'
            string = string + ('+----' * self.n) + '+' + '\n'
        return string
```

```
    def __eq__(self, other):
        if other is None:
            return False

        for i in range(self.n * self.n):
            if self.board[i] != other.board[i]:
                return False
        return True

    def clone(self):
        return Board(self.board.copy(), goalState=self.goalState, move=self.move + 1, previous=self, n=self.n)

    def getBlank(self):
        return self.board.index(0)

    def swap(self, source, destination):
        self.board[source], self.board[destination] = self.board[destination], self.board[source]

    def moveBlank(self, direction):
        blank = self.getBlank()

        if direction == "LEFT":
            if blank % self.n != 0:
                col = (blank % self.n) - 1
                row = int(blank / self.n)
                self.swap(row * self.n + col, blank)

        if direction == "RIGHT":
            if blank % self.n != self.n - 1:
                col = (blank % self.n) + 1
                row = int(blank / self.n)
                self.swap(row * self.n + col, blank)
```

```
    if direction == "UP":
        if int(blank / self.n) != 0:
            col = (blank % self.n)
            row = int(blank / self.n) - 1
            self.swap(row * self.n + col, blank)

    if direction == "DOWN":
        if int(blank / self.n) != self.n - 1:
            col = (blank % self.n)
            row = int(blank / self.n) + 1
            self.swap(row * self.n + col, blank)

def getNeighbours(self):
    blank = self.getBlank()
    neighbours = []

    if blank % self.n != 0:
        newBoard = self.clone()
        newBoard.moveBlank('LEFT')
        neighbours.append(newBoard)

    if blank % self.n != self.n - 1:
        newBoard = self.clone()
        newBoard.moveBlank('RIGHT')
        neighbours.append(newBoard)

    if int(blank / self.n) != 0:
        newBoard = self.clone()
        newBoard.moveBlank('UP')
        neighbours.append(newBoard)

    if int(blank / self.n) != self.n - 1:
        newBoard = self.clone()
        newBoard.moveBlank('DOWN')
```

```
            neighbours.append(newBoard)

        return neighbours

    def isGoal(self):
        for i in range(0, self.n * self.n):
            if i != self.n * self.n - 1:
                if self.board[i] != self.goalState[i]:
                    return False
        return True

    def manhattan(self):
        manhattan = 0

        for i in range(0, self.n * self.n):
            if self.board[i] != self.goalState[i] and self.board[i] != 0:
                position = self.n - 1 if self.board[i] == 0 else self.board[i] - 1
                sRow = int(i / self.n)
                sCol = i % self.n
                dRow = int(position / self.n)
                dCol = position % self.n
                manhattan += abs(sRow - dRow) + abs(sCol - dCol)

        return manhattan

    def getPriority(self, count):
        return self.move + self.manhattan(), count, self

print("Use 0 to denote the space in the board")
solver = PuzzleSolver(n=3)
solver.start(goalState=False)
solver.solveAStart()
solver.solveBFS()
solver.solveDFS()
```


C:\Windows\System32\cmd.e
Microsoft Windows [Version 10.0.26100.3476]
(c) Microsoft Corporation. All rights reserved.
C:\Users\Ayush\Desktop\3317-Ayush\LP-II\AI\2-n-puzzle-using-bfs-dfs-a-star> python code.py < input.txt
Use 0 to denote the space in the board
Start State .....
+----+----+----+
| 01 | 02 | 03 |
+----+----+----+
| 04 | 05 | |
+----+----+----+
| 07 | 08 | 06 |
+----+----+----+
Goal State .....
+----+----+----+
| 01 | 02 | 03 |
+----+----+----+
| 04 | 05 | 06 |
+----+----+----+
| 07 | 08 | |
+----+----+----+
****
Algorithm name :: A star
Total optimal moves to solve :: 1
Total steps required to get to Goal :: 4
Time required to find the Goal state :: 0.0 s
+----+----+----+
| 01 | 02 | 03 |
+----+----+----+
| 04 | 05 | 06 |
+----+----+----+
| 07 | 08 | |
+----+----+----+
****
****
Algorithm name :: BFS
Total optimal moves to solve :: 1
Total steps required to get to Goal :: 9
Time required to find the Goal state :: 0.0 s
+----+----+----+
| 01 | 02 | 03 |
+----+----+----+
| 04 | 05 | 06 |
+----+----+----+
| 07 | 08 | |
+----+----+----+
****
****
Algorithm name :: DFS
Total optimal moves to solve :: 1
Total steps required to get to Goal :: 4
Time required to find the Goal state :: 0.0 s
+----+----+----+
| 01 | 02 | 03 |
+----+----+----+
| 04 | 05 | 06 |
+----+----+----+
| 07 | 08 | |
+----+----+----+
****
C:\Users\Ayush\Desktop\3317-Ayush\LP-II\AI\2-n-puzzle-using-bfs-dfs-a-star>