```
import collections
import queue
import time
import itertools

class Node:

    def __init__(self, puzzle, last=None):
        self.puzzle = puzzle
        self.last = last

    @property
    def seq(self): # to keep track of the sequence used to get to the goal
        node, seq = self, []
        while node:
            seq.append(node)
            node = node.last
        yield from reversed(seq)

    @property
    def state(self):
        return str(self.puzzle.board) # hashable so it can be compared in sets

    @property
    def isSolved(self):
        return self.puzzle.isSolved

    @property
    def getMoves(self):
        return self.puzzle.getMoves

class Puzzle:

    def __init__(self, startBoard):
        self.board = startBoard

    @property
    def getMoves(self):

        possibleNewBoards = []

        zeroPos = self.board.index(0) # find the zero tile to determine possible moves

        if zeroPos == 0:
            possibleNewBoards.append(self.move(0,1))
            possibleNewBoards.append(self.move(0,3))
        elif zeroPos == 1:
            possibleNewBoards.append(self.move(1,0))
            possibleNewBoards.append(self.move(1,2))
            possibleNewBoards.append(self.move(1,4))
        elif zeroPos == 2:
            possibleNewBoards.append(self.move(2,1))
            possibleNewBoards.append(self.move(2,5))
        elif zeroPos == 3:
            possibleNewBoards.append(self.move(3,0))
            possibleNewBoards.append(self.move(3,4))
            possibleNewBoards.append(self.move(3,6))
        elif zeroPos == 4:
            possibleNewBoards.append(self.move(4,1))
            possibleNewBoards.append(self.move(4,3))
            possibleNewBoards.append(self.move(4,5))
            possibleNewBoards.append(self.move(4,7))
        elif zeroPos == 5:
            possibleNewBoards.append(self.move(5,2))
            possibleNewBoards.append(self.move(5,4))
            possibleNewBoards.append(self.move(5,8))
        elif zeroPos == 6:
            possibleNewBoards.append(self.move(6,3))
            possibleNewBoards.append(self.move(6,7))
        elif zeroPos == 7:
            possibleNewBoards.append(self.move(7,4))
            possibleNewBoards.append(self.move(7,6))
            possibleNewBoards.append(self.move(7,8))
        else:
            possibleNewBoards.append(self.move(8,5))
            possibleNewBoards.append(self.move(8,7))

        return possibleNewBoards # returns Puzzle objects (maximum of 4 at a time)

    def move(self, current, to):
```

```
            changeBoard = self.board[:] # create a copy
            changeBoard[to], changeBoard[current] = changeBoard[current], changeBoard[to] # switch the tiles at the passed positions
            return Puzzle(changeBoard) # return a new Puzzle object

        def printPuzzle(self): # prints board in 8 puzzle style

            copyBoard = self.board[:]
            for i in range(9):
                if i == 2 or i == 5:
                    print((str)(copyBoard[i]))
                else:
                    print((str)(copyBoard[i])+" ", end="")
            print('\n')

        @property
        def isSolved(self):
            return self.board == [0,1,2,3,4,5,6,7,8] # goal board

    class Solver:

        def __init__(self, Puzzle):
            self.puzzle = Puzzle

        def IDDFS(self):

            def DLS(currentNode, depth):
                if depth == 0:
                    return None
                if currentNode.isSolved:
                    return currentNode
                elif depth > 0:
                    for board in currentNode.getMoves:
                        nextNode = Node(board, currentNode)
                        if nextNode.state not in visited:
                            visited.add(nextNode.state)
                            goalNode = DLS(nextNode, depth - 1)
                            if goalNode != None: # I thought this should be redundant but it never finds a soln if I take it out
                                if goalNode.isSolved: # same as above ^
                                    return goalNode

            for depth in itertools.count():
                visited = set()
                startNode = Node(self.puzzle)
                print(startNode.isSolved)
                goalNode = DLS(startNode, depth)
                if goalNode != None:
                    if goalNode.isSolved:
                        return goalNode.seq

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

    myPuzzle = Puzzle(startingBoard)
    mySolver = Solver(myPuzzle)
    start = time.time()
    goalSeq = mySolver.IDDFS()
    end = time.time()

    counter = -1 # starting state doesn't count as a move
    for node in goalSeq:
        counter = counter + 1
        node.puzzle.printPuzzle()
    print("Total number of moves: " + str(counter))
    totalTime = end - start
    print("Total searching time: %.2f seconds" % (totalTime))
```

```
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
```

```
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
False
7 2 4
5 0 6
8 3 1

7 0 4
5 2 6
8 3 1

0 7 4
5 2 6
8 3 1

5 7 4
```