**Aim:** **Write a program to solve 8 puzzle problem using A\* algorithm.**

**Code:**

from random import choice

from heapq import heappush, heappop ,heapify

from random import shuffle

import time

class Solver:

def \_\_init\_\_(self, initial\_state=None):

self.initial\_state = State(initial\_state)

self.goal = range(1, 9)

def \_rebuildPath(self, end):

path = [end]

state = end.parent

while state.parent:

path.append(state)

state = state.parent

return path

def solve(self):

openset = PriorityQueue()

openset.add(self.initial\_state)

closed = set()

moves = 0

print 'trying to solve:'

print openset.peek(), '\n\n'

start = time.time()

while openset:

current = openset.poll()

if current.values[:-1] == self.goal:

end = time.time()

print 'I found a solution'

path = self.\_rebuildPath(current)

for state in reversed(path):

print state

print

print 'resolved with% d moves' % len(path)

print 'found the solution in% 2.f seconds' % float(end - start)

break

moves += 1

for state in current.possible\_moves(moves):

if state not in closed:

openset.add(state)

closed.add(current)

else:

print 'I could not solve it!'

class State:

def \_\_init\_\_(self, values, moves=0, parent=None):

self.values = values

self.moves = moves

self.parent = parent

self.goal = range(1, 9)

def possible\_moves(self, moves):

i = self.values.index(0)

if i in [3, 4, 5, 6, 7, 8]:

new\_board = self.values[:]

new\_board[i], new\_board[i - 3] = new\_board[i - 3], new\_board[i]

yield State(new\_board, moves, self)

if i in [1, 2, 4, 5, 7, 8]:

new\_board = self.values[:]

new\_board[i], new\_board[i - 1] = new\_board[i - 1], new\_board[i]

yield State(new\_board, moves, self)

if i in [0, 1, 3, 4, 6, 7]:

new\_board = self.values[:]

new\_board[i], new\_board[i + 1] = new\_board[i + 1], new\_board[i]

yield State(new\_board, moves, self)

if i in [0, 1, 2, 3, 4, 5]:

new\_board = self.values[:]

new\_board[i], new\_board[i + 3] = new\_board[i + 3], new\_board[i]

yield State(new\_board, moves, self)

def score(self):

return self.\_h() + self.\_g()

def \_h(self):

return sum([1 if self.values[i] != self.goal[i] else 0 for i in xrange(8)])

def \_g(self):

return self.moves

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

return self.values == other.values

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

return self.\_\_cmp\_\_(other)

def \_\_hash\_\_(self):

return hash(str(self.values))

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

return self.score() < other.score()

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

return '\n'.join([str(self.values[:3]),

str(self.values[3:6]),

str(self.values[6:9])]).replace('[', '').replace(']', '').replace(',', '').replace('0', 'x')

class PriorityQueue:

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

self.pq = []

def add(self, item):

heappush(self.pq, item)

def poll(self):

return heappop(self.pq)

def peek(self):

return self.pq[0]

def remove(self, item):

value = self.pq.remove(item)

heapify(self.pq)

return value is not None

def \_\_len\_\_(self):

return len(self.pq)

puzzle = range(9)

shuffle(puzzle)

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

solver = Solver(puzzle)

solver.solve()

**Output:**

