In [1]:

```python
import time
import heapq
import copy
```

In [2]:

```python
UNIFORM_COST = 'Uniform cost'
MISPLACED_TILES = 'Misplaced tiles heuristic'
MANHATTAND_DIST = 'Manhattan distance heuristic'
heuristic_choices = {
    '1' : UNIFORM_COST,
    '2' : MISPLACED_TILES,
    '3' : MANHATTAND_DIST
}

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

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

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

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

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

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

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

#                 [[0, 7, 2],
#                  [4, 6, 1],
#                  [3, 5, 8]],
]
```

In [3]:

```python
print(test_cases)
```

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

In [4]:

```python
# The class is used to maintain the state of any orientation during the Eight Puzzle simulation
class EightPuzzleNode:

    def __init__(self, board, heuristic, depth):

        self._board = board
        self._heuristic = heuristic
        self._goal_state = \
                [[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 0]]
        self._goal_value_position_mapping = self._getValuePositionMapping()
        self._blankCellIndex = self._setBlankCellIndex()
        self._depth = depth
        self._calculateTotalCost()


    def getBoard(self):
        return copy.deepcopy(self._board)

    # maps the goal position of a tile with a number from 1-8
    def _getValuePositionMapping(self):

        value_position_mapping = {}
        for row_indx in range(3):
            for col_indx in range(3):

                value_position_mapping[self._goal_state[row_indx][col_indx]] = \
                    (row_indx, col_indx)

        return value_position_mapping

    # returns the depth of the state
    def getDepth(self):

        return self._depth

    # returns the heuristic used
    def getHeuristic(self):

        return self._heuristic

    # sets the blank cell index of the state
    def _setBlankCellIndex(self):

        for row_indx in range(3):

            for col_indx in range(3):

                if self._board[row_indx][col_indx] == 0:
                    return row_indx, col_indx

    # returns the blank cell index of the state
    def getBlankCellIndex(self):

        return self._blankCellIndex

    # checks if the state is in a solved/goal orientation
    def isSolved(self):

        for row_indx in range(3):

            if self._board[row_indx] != self._goal_state[row_indx]:
                return False

        return True

    # prints the current state
    def printCurrentState(self):

        for row_indx in range(3):
            print(self._board[row_indx])

    # calculates the number of misplaced tiles
    def _findMisplacedTilesCount(self):

        misplaced_tiles_count = 0
        for row_indx in range(3):
            for col_indx in range(3):

                if self._board[row_indx][col_indx] and \
                    self._board[row_indx][col_indx] != self._goal_state[row_indx][col_indx]:
                        misplaced_tiles_count+= 1

#         print(misplaced_tiles_count)
        return misplaced_tiles_count

    # calculates the manhattan distance of the current state
    def _findManhattanDistance(self):

        manhattan_distance = 0
        for row_indx in range(3):
            for col_indx in range(3):

                if self._board[row_indx][col_indx] and \
                    self._board[row_indx][col_indx] != self._goal_state[row_indx][col_indx]:

                    goal_row_indx, goal_col_indx = \
                        self._goal_value_position_mapping[ self._board[row_indx][col_indx] ]

                    distance = abs(row_indx - goal_row_indx) + abs(col_indx - goal_col_indx)
                    manhattan_distance += distance

#         print(manhattan_distance)
        return manhattan_distance

    # calculates the total cost according to the heuristic used for the search
    def _calculateTotalCost(self):


        if self._heuristic == UNIFORM_COST:
            self._total_cost = self._depth

        elif self._heuristic ==  MISPLACED_TILES:
            self._total_cost = self._depth + self._findMisplacedTilesCount()

        else:
            self._total_cost = self._depth + self._findManhattanDistance()

    # returns the total cost calculated
    def getTotalCost(self):

        return self._total_cost

    # returns the current cost and heuristic cost
    def getCurrentAndHeuristicCost(self):

        if self._heuristic == UNIFORM_COST:
            return self._depth, 0

        elif self._heuristic ==  MISPLACED_TILES:
            return  self._depth, self._findMisplacedTilesCount()

        else:
            return self._depth, self._findManhattanDistance()

    # comparator function used for the priority queue
    def __lt__(self, nxt):

        if self._heuristic != UNIFORM_COST:
            if self.getTotalCost() == nxt.getTotalCost():
                return self.getDepth() < nxt.getDepth()

        return self.getTotalCost() < nxt.getTotalCost()
```

In [5]:

```python
# gets selection of heuristic to be used from the user
def select_heuristic_search_approach():

    while True:
        heuristic_choice = input('Press 1 to select Uniform Cost Search\n' + \
                                 'Press 2 to select Misplaced Tile Heuristic search\n' + \
                                 'Press 3 to select Manhattan Distance Search\n').strip()

        if heuristic_choice not in heuristic_choices:
            print('Wrong selection')
            continue
        else:
            print(heuristic_choices[heuristic_choice], ' selected')

            return heuristic_choices[heuristic_choice]

# gets input of a new puzzle from the user
def get_new_puzzle():

    print('You will be prompted to enter each row of three rows of the puzzle\n' + \
          'You need to use a zero to represent the blank in your puzzle \n' + \
          'Each of the row can contain three digits delimited by spaces\n' + \
          'Please enter a valid puzzle')

    puzzle = []
    for row_index in range(3):
        row_values = input('Enter row ' + str(row_index + 1) + '\t')
        puzzle.append(list(map(int, row_values.split())))

    return puzzle

# converts the board state to a string to mantain the repeated states of the solution
# search space
def getPuzzleString(board):

    puzzle_string = ''
    for row_indx in range(3):
        for col_indx in range(3):
            puzzle_string += str(board[row_indx][col_indx])

    return puzzle_string

# creates a priority queue
def makeQueue(puzzle):

    nodes = [puzzle]
    heapq.heapify(nodes)

    return nodes

# finds the expanded nodes from the given node
def getNewNode(blank_row_idx, blank_col_idx, board, heuristic_choice, depth,
            new_blank_row_idx, new_blank_col_idx, repeated_states):

    if 0<=new_blank_row_idx<=2 and 0<=new_blank_col_idx<=2:

        new_board = copy.deepcopy(board)
        new_board[blank_row_idx][blank_col_idx], \
            new_board[new_blank_row_idx][new_blank_col_idx] = \
                board[new_blank_row_idx][new_blank_col_idx], \
                    board[blank_row_idx][blank_col_idx]

        new_puzzle_string = getPuzzleString(new_board)
        if new_puzzle_string not in repeated_states:
            return EightPuzzleNode(new_board, heuristic_choice, depth)


    return None

# returns the expanded nodes from the given node
def getNewNodes(puzzle_node, repeated_states):

    blank_row_idx, blank_col_idx = puzzle_node.getBlankCellIndex()
    board = puzzle_node.getBoard()
    heuristic_choice = puzzle_node.getHeuristic()
    depth = puzzle_node.getDepth()

    new_nodes = []

    # left
    new_node = getNewNode(blank_row_idx, blank_col_idx, board, heuristic_choice,
                          depth + 1, blank_row_idx, blank_col_idx - 1,
                         repeated_states)
    if not isinstance(new_node, type(None)):
        new_nodes.append(new_node)

    # right
    new_node = getNewNode(blank_row_idx, blank_col_idx, board, heuristic_choice,
                          depth + 1, blank_row_idx, blank_col_idx + 1,
                          repeated_states)
    if not isinstance(new_node, type(None)):
        new_nodes.append(new_node)

    # top
    new_node = getNewNode(blank_row_idx, blank_col_idx, board, heuristic_choice,
                          depth + 1, blank_row_idx - 1, blank_col_idx,
                          repeated_states)
    if not isinstance(new_node, type(None)):
        new_nodes.append(new_node)

    # bottom
    new_node = getNewNode(blank_row_idx, blank_col_idx, board, heuristic_choice,
                          depth + 1, blank_row_idx + 1, blank_col_idx,
                          repeated_states)

    if not isinstance(new_node, type(None)):
        new_nodes.append(new_node)


    return new_nodes

# returns the solution for the given puzzle node
def solvePuzzle(puzzle_node):

    max_queue_size = 0
    depth = -1
    expanded_node_count = 0
    start_time = time.time()

    repeated_states = {}
    nodes = makeQueue(puzzle_node)
    repeated_states[getPuzzleString(puzzle_node.getBoard())] = True

    while True:

        if len(nodes) > max_queue_size:
            max_queue_size = len(nodes)

        if len(nodes) == 0:
            is_solved = False
            break

        node = heapq.heappop(nodes)
        g_n, h_n = node.getCurrentAndHeuristicCost()
        print('The best state to expand with a g(n) = ', g_n, ' and h(n) = ', h_n)
        node.printCurrentState()

        if node.isSolved():
            depth = node.getDepth()
            is_solved = True
            break
        else:
            new_nodes = getNewNodes(node, repeated_states)
            if len(new_nodes) > 0:
                expanded_node_count += 1

            for new_node in new_nodes:
                heapq.heappush(nodes, new_node)
#                 print(new_node.printCurrentState())


    end_time = time.time()
    execution_time_in_seconds = end_time - start_time

    return (is_solved, max_queue_size, depth,
            expanded_node_count, "{:.1f}s".format(execution_time_in_seconds))
```

In [6]:

```python
# driver function
if __name__ == "__main__":

    while True:

        heuristic_choice = select_heuristic_search_approach()

        input_case = input('Press 1 to run eight puzzle simulation with default test cases\n' + \
                           'Press 2 to enter a new puzzle\n' + \
                           'Press any key to exit\n').strip()

        if input_case == '1':

            counter = 0
            for board in test_cases:
                puzzle = EightPuzzleNode(board, heuristic_choice, 0)

                print()
                print()
                print("Puzzle Test Case: ", counter + 1)
                puzzle.printCurrentState()
                print(heuristic_choice)

                is_solved, max_queue_size, depth, \
                    expanded_node_count, execution_time_in_seconds = \
                                solvePuzzle(puzzle)

                print('is_solved = ', is_solved, ' max_queue_size = ', max_queue_size,
                      ' depth = ', depth, ' expanded_node_count = ', expanded_node_count,
                      ' execution_time_in_seconds = ', execution_time_in_seconds)

                counter += 1


        elif input_case == '2':

            board = get_new_puzzle()
            puzzle = EightPuzzleNode(board, heuristic_choice, 0)

            print("Puzzle Test Case: ", counter + 1)
            puzzle.printCurrentState()
            print(heuristic_choice)

            is_solved, max_queue_size, depth, \
                    expanded_node_count, execution_time_in_seconds = \
                                solvePuzzle(puzzle)

            print('is_solved = ', is_solved, ' max_queue_size = ', max_queue_size,
                  ' depth = ', depth, ' expanded_node_count = ', expanded_node_count,
                  ' execution_time_in_seconds = ', execution_time_in_seconds)

        else:
            print('Thank you')
            break
```

```
Press 1 to select Uniform Cost Search
Press 2 to select Misplaced Tile Heuristic search
Press 3 to select Manhattan Distance Search
3
Manhattan distance heuristic  selected
Press 1 to run eight puzzle simulation with default test cases
Press 2 to enter a new puzzle
Press any key to exit
1


Puzzle Test Case:  1
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Manhattan distance heuristic
The best state to expand with a g(n) =  0  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  1  depth =  0  expanded_node_count =  0  execution_time_in_seconds =  0.0s


Puzzle Test Case:  2
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
Manhattan distance heuristic
The best state to expand with a g(n) =  0  and h(n) =  2
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) =  1  and h(n) =  1
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  3  depth =  2  expanded_node_count =  2  execution_time_in_seconds =  0.0s


Puzzle Test Case:  3
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
Manhattan distance heuristic
The best state to expand with a g(n) =  0  and h(n) =  4
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
The best state to expand with a g(n) =  1  and h(n) =  3
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  2  and h(n) =  2
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  1
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  8  depth =  4  expanded_node_count =  4  execution_time_in_seconds =  0.0s
Press 1 to select Uniform Cost Search
Press 2 to select Misplaced Tile Heuristic search
Press 3 to select Manhattan Distance Search
2
Misplaced tiles heuristic  selected
Press 1 to run eight puzzle simulation with default test cases
Press 2 to enter a new puzzle
Press any key to exit
1


Puzzle Test Case:  1
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Misplaced tiles heuristic
The best state to expand with a g(n) =  0  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  1  depth =  0  expanded_node_count =  0  execution_time_in_seconds =  0.0s


Puzzle Test Case:  2
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
Misplaced tiles heuristic
The best state to expand with a g(n) =  0  and h(n) =  2
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) =  1  and h(n) =  1
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  3  depth =  2  expanded_node_count =  2  execution_time_in_seconds =  0.0s


Puzzle Test Case:  3
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
Misplaced tiles heuristic
The best state to expand with a g(n) =  0  and h(n) =  4
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
The best state to expand with a g(n) =  1  and h(n) =  3
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  2  and h(n) =  2
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  1
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  8  depth =  4  expanded_node_count =  4  execution_time_in_seconds =  0.0s
Press 1 to select Uniform Cost Search
Press 2 to select Misplaced Tile Heuristic search
Press 3 to select Manhattan Distance Search
1
Uniform cost  selected
Press 1 to run eight puzzle simulation with default test cases
Press 2 to enter a new puzzle
Press any key to exit
1


Puzzle Test Case:  1
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Uniform cost
The best state to expand with a g(n) =  0  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  1  depth =  0  expanded_node_count =  0  execution_time_in_seconds =  0.0s


Puzzle Test Case:  2
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
Uniform cost
The best state to expand with a g(n) =  0  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) =  1  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
The best state to expand with a g(n) =  1  and h(n) =  0
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  4  depth =  2  expanded_node_count =  3  execution_time_in_seconds =  0.0s


Puzzle Test Case:  3
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
Uniform cost
The best state to expand with a g(n) =  0  and h(n) =  0
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]
The best state to expand with a g(n) =  1  and h(n) =  0
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  1  and h(n) =  0
[1, 0, 3]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  1  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[4, 0, 8]
The best state to expand with a g(n) =  1  and h(n) =  0
[1, 2, 3]
[5, 6, 0]
[4, 7, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[0, 4, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 2, 0]
[5, 6, 3]
[4, 7, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 2, 3]
[5, 6, 8]
[4, 7, 0]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[4, 8, 0]
The best state to expand with a g(n) =  2  and h(n) =  0
[0, 1, 3]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[1, 3, 0]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  2  and h(n) =  0
[0, 2, 3]
[1, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[5, 6, 0]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 0, 3]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[2, 0, 3]
[1, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 3, 6]
[5, 2, 0]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[0, 7, 6]
[5, 4, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 0, 3]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[4, 0, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[5, 6, 8]
[4, 0, 7]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[5, 6, 0]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[5, 1, 3]
[0, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 0, 2]
[5, 6, 3]
[4, 7, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[4, 0, 8]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[5, 7, 0]
[4, 8, 6]
The best state to expand with a g(n) =  3  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 3, 0]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[0, 4, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[5, 1, 3]
[2, 0, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[4, 8, 0]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 0]
[5, 6, 3]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 6, 8]
[4, 7, 0]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 6, 8]
[4, 7, 0]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 6, 8]
[0, 4, 7]
The best state to expand with a g(n) =  4  and h(n) =  0
[2, 5, 3]
[1, 0, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 3, 6]
[5, 2, 8]
[4, 7, 0]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[0, 4, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[0, 1, 3]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[0, 2, 3]
[1, 7, 6]
[5, 4, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[7, 0, 6]
[5, 4, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 3, 0]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 3, 0]
[5, 2, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 0, 8]
[4, 6, 7]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[4, 8, 0]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 3, 6]
[5, 0, 2]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[0, 2, 3]
[1, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 7, 6]
[4, 8, 0]
The best state to expand with a g(n) =  4  and h(n) =  0
[2, 3, 0]
[1, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[0, 2, 3]
[1, 5, 6]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[5, 6, 8]
[4, 7, 0]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 6, 2]
[5, 0, 3]
[4, 7, 8]
The best state to expand with a g(n) =  4  and h(n) =  0
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
is_solved =  True  max_queue_size =  82  depth =  4  expanded_node_count =  57  execution_time_in_seconds =  0.
0s
Press 1 to select Uniform Cost Search
Press 2 to select Misplaced Tile Heuristic search
Press 3 to select Manhattan Distance Search
2
Misplaced tiles heuristic  selected
Press 1 to run eight puzzle simulation with default test cases
Press 2 to enter a new puzzle
Press any key to exit
k
Thank you
```

In [7]:

```python
#
```

In [ ]:

```python

```