nPuzzle.py

```python
import TreeNode
import heapq as min_heap_esque_queue # because it sort of acts like a min heap

trivial = [[1, 2, 3],
           [4, 5, 6],
           [7, 8, 0]]
veryEasy = [[1, 2, 3],
            [4, 5, 6],
            [7, 0, 8]]
easy = [[1, 2, 0],
        [4, 5, 3],
        [7, 8, 6]]
doable = [[0, 1, 2],
          [4, 5, 3],
          [7, 8, 6]]
oh_boy = [[8, 7, 1],
         [6, 0, 2],
         [5, 4, 3]]
impossible = [[1, 2, 3],
              [4, 5, 6],
              [8, 7, 0]]

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


def main():
    puzzle_mode = input("Welcome to an 8-Puzzle Solver. Type '1' to use a default puzzle, or '2' to create your own."
                        + '\n')
    if puzzle_mode == "1":
        select_and_init_algorithm(init_default_puzzle_mode())

    if puzzle_mode == "2":
        print("Enter your puzzle, using a zero to represent the blank. " +
              "Please only enter valid 8-puzzles. Enter the puzzle demilimiting " +
              "the numbers with a space. RET only when finished." + '\n')
        puzzle_row_one = input("Enter the first row: ")
        puzzle_row_two = input("Enter the second row: ")
```

```
        puzzle_row_three = input("Enter␣the␣third␣row:␣")

        puzzle_row_one = puzzle_row_one.split()
        puzzle_row_two = puzzle_row_two.split()
        puzzle_row_three = puzzle_row_three.split()

        for i in range(0, 3):
            puzzle_row_one[i] = int(puzzle_row_one[i])
            puzzle_row_two[i] = int(puzzle_row_two[i])
            puzzle_row_three[i] = int(puzzle_row_three[i])

        user_puzzle = [puzzle_row_one, puzzle_row_two,
            ↪ puzzle_row_three]
        select_and_init_algorithm(user_puzzle)

    return


def init_default_puzzle_mode():
    selected_difficulty = input(
        "You␣wish␣to␣use␣a␣default␣puzzle.␣Please␣enter␣a␣desired␣
            ↪ difficulty␣on␣a␣scale␣from␣0␣to␣5." + '\n')
    if selected_difficulty == "0":
        print("Difficulty␣of␣'Trivial'␣selected.")
        return trivial
    if selected_difficulty == "1":
        print("Difficulty␣of␣'Very␣Easy'␣selected.")
        return veryEasy
    if selected_difficulty == "2":
        print("Difficulty␣of␣'Easy'␣selected.")
        return easy
    if selected_difficulty == "3":
        print("Difficulty␣of␣'Doable'␣selected.")
        return doable
    if selected_difficulty == "4":
        print("Difficulty␣of␣'Oh␣Boy'␣selected.")
        return oh_boy
    if selected_difficulty == "5":
        print("Difficulty␣of␣'Impossible'␣selected.")
        return impossible


def print_puzzle(puzzle):
    for i in range(0, 3):
        print(puzzle[i])
    print('\n')
```

```
def select_and_init_algorithm(puzzle):
    algorithm = input("Select␣algorithm.␣(1)␣for␣Uniform␣Cost␣
        ↪ Search,␣(2)␣for␣the␣Misplaced␣Tile␣Heuristic,␣"
                      "or␣(3)␣the␣Manhattan␣Distance␣Heuristic." +
                          ↪ '\n')
    if algorithm == "1":
        uniform_cost_search(puzzle, 0)
    if algorithm == "2":
        uniform_cost_search(puzzle, 1)
    if algorithm == "3":
        uniform_cost_search(puzzle, 2)


def uniform_cost_search(puzzle, heuristic):

    starting_node = TreeNode.TreeNode(None, puzzle, 0, 0)
    working_queue = []
    repeated_states = dict()
    min_heap_esque_queue.heappush(working_queue, starting_node)
    num_nodes_expanded = 0
    max_queue_size = 0
    repeated_states[starting_node.board_to_tuple()] = "This␣is␣the
        ↪ ␣parent␣board"

    stack_to_print = [] # the board states are stored in a stack

    while len(working_queue) > 0:
        max_queue_size = max(len(working_queue), max_queue_size)
        # the node from the queue being considered/checked
        node_from_queue = min_heap_esque_queue.heappop(
            ↪ working_queue)
        repeated_states[node_from_queue.board_to_tuple()] = "This␣
            ↪ can␣be␣anything"
        if node_from_queue.solved(): # check if the current state
            ↪ of the board is the solution
            while len(stack_to_print) > 0: # the stack of nodes
                ↪ for the traceback
                print_puzzle(stack_to_print.pop())
            print("Number␣of␣nodes␣expanded:", num_nodes_expanded)
            print("Max␣queue␣size:", max_queue_size)
            return node_from_queue

        stack_to_print.append(node_from_queue.board)
        # expand children : children_from_node is a list of
```

```
        expanded children's nodes
        children_from_node = node_from_queue.expand_children(heuristic)
        # push non-duplicate children to working_queue
        for expanded_child in children_from_node:
            if expanded_child.board_to_tuple() not in repeated_states:
                min_heap_esque_queue.heappush(working_queue, expanded_child)
                num_nodes_expanded += 1
            # Hash in tuples
            repeated_states[expanded_child.board_to_tuple()] = "This the newest unique board of an expanded child"

    if len(working_queue) == 0:
        print(num_nodes_expanded)
        print(max_queue_size)
        print("Failure. No solution.")

    return


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

TreeNode.py

```
import copy

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

# a matrix of distances, matrix[i][j] = manhattan distance from i to j
manhattan_distance_matrix = [[0, 1, 2, 1, 2, 3, 2, 3],
                             [1, 0, 1, 2, 1, 2, 3, 2],
                             [2, 1, 0, 3, 2, 1, 4, 3],
                             [1, 2, 3, 0, 1, 2, 1, 2],
                             [2, 1, 2, 1, 0, 1, 2, 1],
                             [3, 2, 1, 2, 1, 0, 1, 2],
                             [2, 3, 4, 1, 2, 1, 0, 1],
                             [3, 2, 3, 2, 1, 2, 1, 0]]
```

```
class TreeNode:

    def __init__(self, parent_node, board, h_n, g_n):

        self.board = board
        self.parent = parent_node
        self.g_n = g_n
        self.h_n = h_n
        return

    def expand_children(self, heuristic):

        if heuristic == 0:
            g_n = 0
        elif heuristic == 1:
            g_n = self.find_misplaced_distance()
        elif heuristic == 2:
            g_n = self.find_manhattan_distance_heuristic()

        # viable moves
        children = [] # a list of boards
        z = self.zero_position() # position of the zero in the
            ↪ parent
        # the following if statements determine the new position
            ↪ of the 0 in the child node
        if z[1] in range(0, 2):
            # can move right
            # c_node is the new child node
            # parameters passed in are the new z position
                ↪ coordinates
            c_right_node_board = self.child_node(z[0], z[1] + 1)
            c_right_node = TreeNode(self, c_right_node_board, self
                ↪ .h_n + 1, g_n)
            children.append(c_right_node)
        if z[1] in range(1, 3):
            # can move left
            c_left_node_board = self.child_node(z[0], z[1] - 1)
            c_left_node = TreeNode(self, c_left_node_board, self.
                ↪ h_n + 1, g_n)
            children.append(c_left_node)
        if z[0] in range(0, 2):
            # can move down
            c_down_node_board = self.child_node(z[0] + 1, z[1])
            c_down_node = TreeNode(self, c_down_node_board, self.
                ↪ h_n + 1, g_n)
            children.append(c_down_node)
```

```
    if z[0] in range(1, 3):
        # can move up
        c_up_node_board = self.child_node(z[0] - 1, z[1])
        c_up_node = TreeNode(self, c_up_node_board, self.h_n +
            ↪  1, g_n)
        children.append(c_up_node)
    return children

def zero_position(self):
    for i in range(3):
        for j in range(3):
            if self.board[i][j] == 0:
                return [i, j]

def __lt__(self, other): # to tell the priority queue how to
    ↪ queue
    return (self.h_n + self.g_n) < (other.h_n + other.g_n)

def child_node(self, y_val, x_val):
    # a copy of the board
    board_copy = copy.deepcopy(self.board)
    # on the parent board: x and y position values of the tile
        ↪  0 is being swapped with
    swapped_val = board_copy[y_val][x_val]
    board_copy[y_val][x_val] = 0
    # set parent 0 position to the swapped value
    board_copy[self.zero_position()[0]][self.zero_position()
        ↪ [1]] = swapped_val

    return board_copy

def board_to_tuple(self):
    return tuple(self.board[0]), tuple(self.board[1]), tuple(
        ↪ self.board[2])

def solved(self):
    return self.board == eight_goal_state

def find_misplaced_distance(self):
    # take board indexes, check against goal state, (ignore 0s
        ↪ )
    # if they don't match, then increment misplaced_distance
    misplaced_distance = 0
    for i in range(0, 3):
        for j in range(0, 3):
            if self.board[i][j] != 0 and (self.board[i][j] !=
```

```
                ↪ eight_goal_state[i][j]):
                misplaced_distance += 1
        self.g_n = misplaced_distance
        return misplaced_distance

    def find_manhattan_distance_heuristic(self):
        manhattan_distance = 0
        for m in range(0, 3):
            for n in range(0, 3):
                if self.board[m][n] != 0 and (self.board[m][n] !=
                    ↪ eight_goal_state[m][n]):
                    manhattan_distance += manhattan_distance_matrix
                        ↪ [self.board[m][n] - 1][eight_goal_state[
                        ↪ m][n] - 1]
        return manhattan_distance
```