Exp: 4: Solve 8-puzzle problem using Best First Search.

Program:

```
import copy
import heapq

class PuzzleNode:
    def __init__(self, state, parent=None, action=None, cost=0,
heuristic=0):
        self.state = state
        self.parent = parent
        self.action = action
        self.cost = cost
        self.heuristic = heuristic
        self.total_cost = cost + heuristic

    def __lt__(self, other):
        return self.total_cost < other.total_cost

    def __eq__(self, other):
        return self.state == other.state

    def __hash__(self):
        return hash(tuple(map(tuple, self.state)))

    def is_goal(self, goal_state):
        return self.state == goal_state

    def generate_children(self):
        x, y = self.find_blank(self.state)
        possible_moves = [(x, y - 1), (x, y + 1), (x - 1, y), (x + 1,
y)]
        children = []

        for move in possible_moves:
            child_state = self.make_move(self.state, (x, y), move)
            if child_state is not None:
                child = PuzzleNode(child_state, parent=self,
action=move, cost=self.cost + 1,
heuristic=self.calculate_heuristic(child_state))
                children.append(child)

        return children

    def find_blank(self, state):
        for i in range(3):
            for j in range(3):
```

```
                if state[i][j] == 0:
                    return i, j

    def make_move(self, state, current_pos, new_pos):
        x1, y1 = current_pos
        x2, y2 = new_pos

        if 0 <= x2 < 3 and 0 <= y2 < 3:
            new_state = copy.deepcopy(state)
            new_state[x1][y1], new_state[x2][y2] = new_state[x2][y2],
new_state[x1][y1]
            return new_state
        else:
            return None

    def calculate_heuristic(self, state):
        # Manhatten distance heuristic
        distance = 0
        for i in range(3):
            for j in range(3):
                value = state[i][j]
                if value != 0:
                    goal_x, goal_y = divmod(value - 1, 3)
                    distance += abs(i - goal_x) + abs(j - goal_y)
        return distance

def best_first_search(initial_state, goal_state):
    start_node = PuzzleNode(initial_state, None, None, 0, 0)
    goal_node = PuzzleNode(goal_state)

    open_set = [start_node]
    closed_set = set()

    while open_set:
        current_node = heapq.heappop(open_set)

        if current_node.is_goal(goal_state):
            return reconstruct_path(current_node)

        closed_set.add(current_node)

        children = current_node.generate_children()
        for child in children:
            if child not in closed_set and child not in open_set:
                heapq.heappush(open_set, child)

    return None
```

```python
def reconstruct_path(node):
    path = []
    while node.parent is not None:
        path.insert(0, (node.action, node.state))
        node = node.parent
    return path

def print_puzzle(puzzle):
    for row in puzzle:
        print(" ".join(map(str, row)))
    print()

def main():
    initial_state = [[2, 8, 3],
                     [1, 6, 4],
                     [7, 0, 5]]

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

    print("Initial State:")
    print_puzzle(initial_state)

    print("Goal State:")
    print_puzzle(goal_state)

    solution_path = best_first_search(initial_state, goal_state)

    if solution_path:
        print("Solution:")
        for step, state in solution_path:
            print(f"Move {step}:")
            print_puzzle(state)
    else:
        print("No solution found.")

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

Output:

```
Initial State:
2 8 3
1 6 4
7 0 5

Goal State:
1 2 3
8 0 4
```

```
7 6 5

Solution:
Move (1, 1):
2 8 3
1 0 4
7 6 5

Move (0, 1):
2 0 3
1 8 4
7 6 5

Move (0, 0):
0 2 3
1 8 4
7 6 5

Move (1, 0):
1 2 3
0 8 4
7 6 5

Move (1, 1):
1 2 3
8 0 4
7 6 5
```

Program:

```
import copy
import heapq

class PuzzleNode:
    def __init__(self, state, parent=None, action=None, cost=0,
heuristic=0):
        self.state = state
        self.parent = parent
        self.action = action
        self.cost = cost
        self.heuristic = heuristic
        self.total_cost = cost + heuristic

    def __lt__(self, other):
        return self.total_cost < other.total_cost

    def __eq__(self, other):
        return self.state == other.state

    def __hash__(self):
        return hash(tuple(map(tuple, self.state)))
```

```
    def is_goal(self, goal_state):
        return self.state == goal_state

    def generate_children(self):
        x, y = self.find_blank(self.state)
        possible_moves = [(x, y - 1), (x, y + 1), (x - 1, y), (x + 1,
y)]
        children = []

        for move in possible_moves:
            child_state = self.make_move(self.state, (x, y), move)
            if child_state is not None:
                child = PuzzleNode(child_state, parent=self,
action=move, cost=self.cost + 1,
heuristic=self.calculate_heuristic(child_state))
                children.append(child)

        return children

    def find_blank(self, state):
        for i in range(3):
            for j in range(3):
                if state[i][j] == 0:
                    return i, j

    def make_move(self, state, current_pos, new_pos):
        x1, y1 = current_pos
        x2, y2 = new_pos

        if 0 <= x2 < 3 and 0 <= y2 < 3:
            new_state = copy.deepcopy(state)
            new_state[x1][y1], new_state[x2][y2] = new_state[x2][y2],
new_state[x1][y1]
            return new_state
        else:
            return None

    def calculate_heuristic(self, state):
        # Manhatten distance heuristic
        distance = 0
        for i in range(3):
            for j in range(3):
                value = state[i][j]
                if value != 0:
                    goal_x, goal_y = divmod(value - 1, 3)
                    distance += abs(i - goal_x) + abs(j - goal_y)
        return distance
```

```python
def best_first_search(initial_state, goal_state):
    start_node = PuzzleNode(initial_state, None, None, 0, 0)
    goal_node = PuzzleNode(goal_state)

    open_set = [start_node]
    closed_set = set()

    while open_set:
        current_node = heapq.heappop(open_set)

        if current_node.is_goal(goal_state):
            return reconstruct_path(current_node)

        closed_set.add(current_node)

        children = current_node.generate_children()
        for child in children:
            if child not in closed_set and child not in open_set:
                heapq.heappush(open_set, child)

    return None

def reconstruct_path(node):
    path = []
    while node.parent is not None:
        path.insert(0, (node.action, node.state))
        node = node.parent
    return path

def print_puzzle(puzzle):
    for row in puzzle:
        print(" ".join(map(str, row)))
    print()

def main():
    initial_state = [[2, 8, 3],
                     [1, 6, 4],
                     [7, 0, 5]]

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

    print("Initial State:")
    print_puzzle(initial_state)

    print("Goal State:")
    print_puzzle(goal_state)
```

```python
    solution_path = best_first_search(initial_state, goal_state)

    if solution_path:
        print("Solution:")
        total_moves = len(solution_path) - 1
        for step, state in solution_path:
            print(f"Move {step}:")
            print_puzzle(state)
        print(f"Total number of moves: {total_moves}")
    else:
        print("No solution found.")

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

```
Initial State:
2 8 3
1 6 4
7 0 5

Goal State:
1 2 3
8 0 4
7 6 5

Solution:
Move (1, 1):
2 8 3
1 0 4
7 6 5

Move (0, 1):
2 0 3
1 8 4
7 6 5

Move (0, 0):
0 2 3
1 8 4
7 6 5

Move (1, 0):
1 2 3
0 8 4
7 6 5

Move (1, 1):
1 2 3
8 0 4
7 6 5

Total number of moves: 4
```

Explanation:

This code implements a Best-First Search algorithm to solve the 8-puzzle problem. The 8-puzzle is a sliding puzzle that consists of a 3x3 grid with eight numbered tiles and an empty space. The goal is to arrange the tiles in a specific order by sliding them into the empty space.

Let's break down the code:

PuzzleNode Class:

__init__: Initializes a PuzzleNode with the current state, parent node, action taken to reach this state, cost, and heuristic value.

__lt__: Compares nodes based on their total cost (cost + heuristic) for priority queue ordering.

__eq__: Checks if two nodes have the same state.

__hash__: Generates a hash based on the state for set operations.

is_goal: Checks if the current node's state is the goal state.

generate_children: Generates child nodes by exploring possible moves.

find_blank: Finds the coordinates of the blank (0) in the puzzle.

make_move: Makes a move by swapping the blank with an adjacent tile.

calculate_heuristic: Calculates the Manhattan distance heuristic.

Best-First Search Function (best_first_search):

Takes the initial state and goal state as input.

Initializes start and goal nodes.

Uses a priority queue (open_set) to explore nodes in order of their total cost.

Uses a set (closed_set) to keep track of visited nodes.

Continues searching until the goal state is reached or no solution is found.

Reconstruct Path Function (reconstruct_path):

Reconstructs the path from the goal node to the start node.

Print Puzzle Function (print_puzzle):

Prints the puzzle in a readable format.

Main Function (main):

Defines an initial state and a goal state for the 8-puzzle.

Prints the initial and goal states.

Calls best_first_search to find a solution path.

If a solution is found, it prints the sequence of moves and corresponding states; otherwise, it indicates that no solution was found.

Execution (`if name == "main":):

Calls the main function when the script is executed.

The code uses the Manhattan distance heuristic and a priority queue (heapq) to efficiently explore the state space. It's a clear and modular implementation of the Best-First Search algorithm for solving the 8-puzzle problem.