ASSIGNMENT 1

Name: Sunbla Khan

ID: SP20-BSSE-0027

Section: AM

1) DFS implementation:

```
import pygame
import random
from collections import deque

# Pygame Setup
pygame.init()
width, height = 300, 300
win = pygame.display.set_mode((width, height))
pygame.display.set_caption("8-Puzzle")

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GREEN = (0, 128, 0)
RED = (255, 0, 0)

font = pygame.font.SysFont(None, 40)

class Puzzle:
    def __init__(self, board=None):
        if board:
            self.board = board
        else:
            self.board = list(range(1, 9)) + [None]
            random.shuffle(self.board)
        self.empty_pos = self.board.index(None)

    def __str__(self):
        return '\n'.join([' '.join(map(str, self.board[i:i+3])) for i in
range(0, 9, 3)])

    def move(self, direction):
        x, y = self.empty_pos % 3, self.empty_pos // 3
        if direction == "left" and x > 0:
            target = self.empty_pos - 1
        elif direction == "right" and x < 2:
            target = self.empty_pos + 1
```

```
        elif direction == "up" and y > 0:
            target = self.empty_pos - 3
        elif direction == "down" and y < 2:
            target = self.empty_pos + 3
        else:
            return None

        new_board = self.board[:]
        new_board[self.empty_pos], new_board[target] = new_board[target],
new_board[self.empty_pos]
        return Puzzle(new_board)

    def solved(self):
        return self.board == list(range(1, 9)) + [None]

    def serialize(self):
        return ''.join(map(str, self.board))

def bfs_solver(initial_state):
    queue = deque([initial_state])
    visited = set([initial_state.serialize()])
    prev_states = {initial_state.serialize(): None}
    actions = {initial_state.serialize(): None}

    while queue:
        current_state = queue.pop()

        if current_state.solved():
            moves = []
            while current_state:
                move = actions[current_state.serialize()]
                if move is not None:
                    moves.append(move)
                current_state = prev_states[current_state.serialize()]
            moves.reverse()
            return moves

        for direction in ["left", "right", "up", "down"]:
            new_state = current_state.move(direction)
            if new_state and new_state.serialize() not in visited:
                visited.add(new_state.serialize())
                queue.append(new_state)
                prev_states[new_state.serialize()] = current_state
                actions[new_state.serialize()] = direction

    return []

def draw_puzzle(puzzle):
    win.fill(WHITE)
    for i, num in enumerate(puzzle.board):
        x, y = (i % 3) * 100, (i // 3) * 100
        if num:
            pygame.draw.rect(win, GREEN, (x, y, 100, 100))
            label = font.render(str(num), True, BLACK)
            win.blit(label, (x + 40, y + 40))
    pygame.display.flip()
```

```
def main():
    puzzle = Puzzle()
    solution = []

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_s:
                    solution = bfs_solver(puzzle)
                if solution:
                    move = solution.pop(0)
                    puzzle = puzzle.move(move)

        draw_puzzle(puzzle)
        pygame.time.wait(500)

    pygame.quit()

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

Output:

2) Heuristic Code:

a) Sum of Misplaced tiles:

```
def sum(a):
    sum = 0
    for i in range(8):
        if(a[i] != i+1):
            sum +=1
    return sum

print(sum([2,3,1,4,5,6,7,8]))
```

Output:

```
C:\Python312\python.exe C:\Users\Usama\PycharmProjects\Assignment1\main.py
3

Process finished with exit code 0
```

b) Manhattan Distance of misplaced tiles:

```
def manhattan_distance(puzzle):
    distance = 0
    for i in range(3):
        for j in range(3):
            if puzzle[i][j] != 0:  # Skip the empty tile
                target_value = i * 3 + j + 1
                target_row = (target_value - 1) // 3
                target_col = (target_value - 1) % 3
                distance += abs(i - target_row) + abs(j - target_col)
    return distance

puzzle = [[2, 3, 1]
    ,     [4, 5, 6],
          [7, 8, 0]]
print(manhattan_distance(puzzle))
```

Output:

```
C:\Python312\python.exe C:\Users\Usama\PycharmProjects\Assignment1\main.py
0

Process finished with exit code 0
```

3) A* implementation:

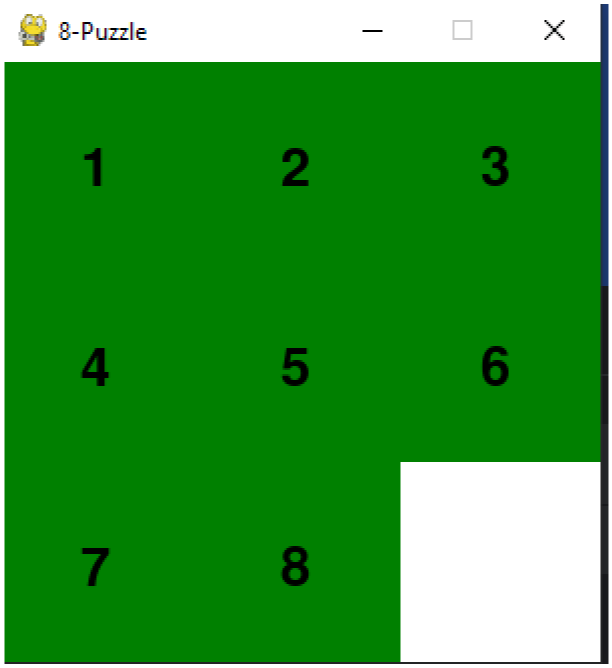
```
import pygame
import random
import heapq

# Pygame Setup
pygame.init()
width, height = 300, 300
win = pygame.display.set_mode((width, height))
pygame.display.set_caption("8-Puzzle")

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GREEN = (0, 128, 0)
RED = (255, 0, 0)

font = pygame.font.SysFont(None, 40)

class Puzzle:
    def __init__(self, board=None):
        if board:
            self.board = board
        else:
            self.board = list(range(1, 9)) + [None]
            random.shuffle(self.board)
        self.empty_pos = self.board.index(None)

    def __str__(self):
        return '\n'.join([' '.join(map(str, self.board[i:i+3])) for i in
range(0, 9, 3)])

    def move(self, direction):
        x, y = self.empty_pos % 3, self.empty_pos // 3
        if direction == "left" and x > 0:
```

```
            target = self.empty_pos - 1
        elif direction == "right" and x < 2:
            target = self.empty_pos + 1
        elif direction == "up" and y > 0:
            target = self.empty_pos - 3
        elif direction == "down" and y < 2:
            target = self.empty_pos + 3
        else:
            return None

        new_board = self.board[:]
        new_board[self.empty_pos], new_board[target] = new_board[target],
new_board[self.empty_pos]
        return Puzzle(new_board)

    def solved(self):
        return self.board == list(range(1, 9)) + [None]

    def serialize(self):
        return ''.join(map(str, self.board))

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

def heuristic_cost(state):
    h = 0
    for i in range(8):
        if state.board[i] is not None:
            x, y = divmod(i, 3)
            target = state.board[i] - 1
            tx, ty = divmod(target, 3)
            h += abs(x - tx) + abs(y - ty)
    return h

def astar_solver(initial_state):
    open_set = [(0, initial_state)]
    closed_set = set()

    g_scores = {initial_state.serialize(): 0}
    f_scores = {initial_state.serialize(): heuristic_cost(initial_state)}

    prev_states = {initial_state.serialize(): None}
    actions = {initial_state.serialize(): None}

    while open_set:
        _, current_state = heapq.heappop(open_set)

        if current_state.solved():
            moves = []
            while current_state:
                move = actions[current_state.serialize()]
                if move is not None:
                    moves.append(move)
                current_state = prev_states[current_state.serialize()]
            moves.reverse()
            return moves
```

```python
        serialized_state = current_state.serialize()

        if serialized_state in closed_set:
            continue

        closed_set.add(serialized_state)

        for direction in ["left", "right", "up", "down"]:
            new_state = current_state.move(direction)
            if new_state:
                new_serialized_state = new_state.serialize()
                tentative_g_score = g_scores[serialized_state] + 1

                if new_serialized_state not in g_scores or tentative_g_score <
g_scores[new_serialized_state]:
                    g_scores[new_serialized_state] = tentative_g_score
                    f_scores[new_serialized_state] = tentative_g_score +
heuristic_cost(new_state)
                    heapq.heappush(open_set, (f_scores[new_serialized_state],
new_state))
                    prev_states[new_serialized_state] = current_state
                    actions[new_serialized_state] = direction

    return []

def draw_puzzle(puzzle):
    win.fill(WHITE)
    for i, num in enumerate(puzzle.board):
        x, y = (i % 3) * 100, (i // 3) * 100
        if num:
            pygame.draw.rect(win, GREEN, (x, y, 100, 100))
            label = font.render(str(num), True, BLACK)
            win.blit(label, (x + 40, y + 40))
    pygame.display.flip()

def main():
    puzzle = Puzzle()
    solution = []

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_s:
                    solution = astar_solver(puzzle)
                if solution:
                    move = solution.pop(0)
                    puzzle = puzzle.move(move)

        draw_puzzle(puzzle)
        pygame.time.wait(500)

    pygame.quit()
```

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

Output:



4) Comparison between the both heuristics :

The comparison between the Sum of Misplaced Tiles heuristic and the Manhattan Distance heuristic, along with their time complexities, is important when choosing a heuristic for solving problems like the 8-Puzzle using search algorithms like A*.

1. **Sum of Misplaced Tiles (Misplaced Tiles Heuristic)**:
   - This heuristic counts the number of tiles that are not in their goal position.
   - It is admissible, which means it never overestimates the cost to reach the goal.

- It's relatively easy to compute since you only need to check each tile's position.
   - However, it might not always provide a good estimate of the actual cost, especially for problems where moving a single tile results in a high cost.
   - The time complexity of this heuristic is O(n), where n is the number of tiles. It requires a linear scan of the puzzle state.

2. **Manhattan Distance (Manhattan Heuristic)**:
   - The Manhattan Distance heuristic calculates the sum of the Manhattan distances (or taxi-cab distances) of each tile from its current position to its goal position.
   - It is also admissible and, in many cases, provides a better estimate of the actual cost compared to the Misplaced Tiles heuristic.
   - It is more complex to compute than the Misplaced Tiles heuristic since it considers the distance each tile needs to travel.
   - The time complexity of this heuristic is O(n), similar to the Misplaced Tiles heuristic, as it also requires a linear scan of the puzzle state. However, the Manhattan Distance involves more arithmetic operations.

In terms of comparison:

- The Manhattan Distance heuristic is generally more informed and provides a better estimate of the actual cost to reach the goal state. It takes into account the distance each tile needs to move, making it more accurate for problems like the 8-Puzzle.

- The Sum of Misplaced Tiles heuristic is simpler to compute and may be sufficient for relatively easy problems. However, it tends to be less accurate and can be less effective for problems with complex state spaces.

When choosing a heuristic for A* search, it's often a good practice to start with the Manhattan Distance heuristic due to its better estimation properties. However, depending on the problem, you may experiment with different heuristics to find the one that works best in practice.

In terms of time complexity, both heuristics have the same O(n) time complexity for evaluating a state. The computational cost mainly depends on the size and complexity of the state space rather than the heuristic used.