تمرین ۲ – مازندرانیان – ۸۳۰۴۰۲۰۶۶

AStar

```
import heapq

def a_star(initial_state, goal_state):
    def get_successors(state):
        from copy import deepcopy
        for i in range(3):
            for j in range(3):
                if state[i][j] == 0:
                    empty_pos = (i, j)
                    break
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        successors = []
        for move in moves:
            new_i, new_j = empty_pos[0] + move[0], empty_pos[1] + move[1]
            if 0 <= new_i < 3 and 0 <= new_j < 3:
                new_state = deepcopy(state)
                new_state[empty_pos[0]][empty_pos[1]], new_state[new_i][new_j] =
new_state[new_i][new_j], new_state[empty_pos[0]][empty_pos[1]]
                successors.append(new_state)
        return successors

    def manhattan_distance(state, goal_state):
        distance = 0
        for i in range(3):
            for j in range(3):
                if state[i][j] != 0:
                    value = state[i][j]
                    goal_i, goal_j = [(index // 3, index % 3) for index, num in
enumerate(sum(goal_state, [])) if num == value][0]
                    distance += abs(i - goal_i) + abs(j - goal_j)
        return distance

    def print_state(state, step):
        print(f"Step {step}:")
        for row in state:
            print(row)
        print("-")

    frontier = []
    heapq.heappush(frontier, (0, initial_state, 0, []))
    visited = set()

    step = 0
    while frontier:
        f, current_state, g, path = heapq.heappop(frontier)

        print_state(current_state, step)
        step += 1

        if current_state == goal_state:
            path.append(current_state)
            print("Found Path:")
            for i, state in enumerate(path):
                print_state(state, i)
            return g

        state_tuple = tuple(tuple(row) for row in current_state)
        if state_tuple in visited:
            continue
        visited.add(state_tuple)

        for successor in get_successors(current_state):
            if tuple(tuple(row) for row in successor) not in visited:
                g_new = g + 1
                h_new = manhattan_distance(successor, goal_state)
                f_new = g_new + h_new
                heapq.heappush(frontier, (f_new, successor, g_new, path + [current_state]))

    return -1

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

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

result = a_star(initial_state, goal_state)
print("Minimum moves to reach goal:", result)
```

کد از الگوریتم A* برای حل پازل ۸ استفاده می‌کند. ابتدا تابع get_successors تمام حالات ممکن بعد از جابجایی خانه خالی را تولید می‌کند. تابع manhattan_distance برای هر حالت مقدار تابع هزینه تخمینی را محاسبه می‌کند. تابع print_state در هر مرحله وضعیت کنونی را نمایش می‌دهد. الگوریتم با استفاده از یک صف اولویت‌دار (heapq) اجرا می‌شود و در هر مرحله گره‌ای با کمترین هزینه را انتخاب می‌کند. اگر گره انتخابی همان وضعیت هدف باشد، مسیر حل نمایش داده می‌شود. در غیر این صورت، وضعیت فعلی به مجموعه بازدیدشده‌ها اضافه شده و همسایه‌های آن بررسی می‌شوند. برای هر همسایه، مقدار جدید g و h محاسبه شده و وضعیت در صف قرار می‌گیرد. در نهایت، تعداد حداقل حرکات موردنیاز برای رسیدن به وضعیت هدف چاپ می‌شود.

Efficient AStar

```python
import heapq

class PuzzleNode:
    def __init__(self, state, parent=None, g=0):
        self.state = state
        self.parent = parent
        self.g = g
        self.h = improved_heuristic(state, goal_state)
        self.f = self.g + self.h

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

    def get_neighbors(self):
        neighbors = []
        empty_x, empty_y = [(i, j) for i in range(3) for j in range(3) if self.state[i][j] ==
0][0]
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

        for dx, dy in moves:
            new_x, new_y = empty_x + dx, empty_y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_state = [row[:] for row in self.state]
                new_state[empty_x][empty_y], new_state[new_x][new_y] = new_state[new_x]
[new_y], new_state[empty_x][empty_y]
                neighbors.append(PuzzleNode(new_state, self, self.g + 1))
        return neighbors


def a_star(start_state, goal_state):
    open_list = []
    closed_set = set()

    start_node = PuzzleNode(start_state)
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node.state == goal_state:
            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            return path[::-1]

        closed_set.add(tuple(map(tuple, current_node.state)))

        for neighbor in current_node.get_neighbors():
            if tuple(map(tuple, neighbor.state)) in closed_set:
                continue
            heapq.heappush(open_list, neighbor)

    return None


def manhattan_distance(state, goal_state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                goal_x, goal_y = [(x, y) for x in range(3) for y in range(3) if goal_state[x]
[y] == value][0]
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance


def linear_conflicts(state, goal_state):
    conflicts = 0
    for row in range(3):
        state_row = state[row]
        goal_row = goal_state[row]
        goal_positions = [goal_row.index(value) for value in state_row if value in goal_row
and value != 0]
        for i in range(len(goal_positions)):
            for j in range(i + 1, len(goal_positions)):
                if goal_positions[i] > goal_positions[j]:
                    conflicts += 1

    for col in range(3):
        state_column = [state[row][col] for row in range(3) if state[row][col] != 0]
        goal_column = [goal_state[row][col] for row in range(3)]
        goal_positions = [goal_column.index(value) for value in state_column if value in
goal_column]
        for i in range(len(goal_positions)):
            for j in range(i + 1, len(goal_positions)):
                if goal_positions[i] > goal_positions[j]:
                    conflicts += 1

    return conflicts * 2


def improved_heuristic(state, goal_state):
    return manhattan_distance(state, goal_state) + linear_conflicts(state, goal_state)


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

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

solution = a_star(initial_state, goal_state)

if solution:
    print("\nSolution Found in {} steps:".format(len(solution) - 1))
    for step, state in enumerate(solution):
        print("\nStep", step)
        for row in state:
            print(row)
else:
    print("No solution found!")
```

این کد از الگوریتم *A برای حل پازل ۸ استفاده می‌کند و از یک معیار بهینه‌تر شامل مجموع فاصله مانهتن و تضادهای خطی استفاده می‌نماید PuzzleNode .هر وضعیت را همراه با والد، هزینه و مقدار تابع ارزیابی نگه می‌دارد. تابع get_neighborsحرکت‌های ممکن را تولید می‌کند. تابع linear_conflictsکاشی‌هایی را که مسیر یکدیگر را مسدود کرده‌اند شناسایی کرده و جریمه‌ای به تابع ارزیابی اضافه می‌کند. الگوریتم با استفاده از صف اولویت‌دار (open_list) وضعیت‌های بهینه‌تر را زودتر بررسی کرده و از مجموعه closed_setبرای جلوگیری از تکرار وضعیت‌ها استفاده می‌کند. پس از رسیدن به حالت هدف، مسیر با استفاده از والدهای هر گره بازسازی شده و نمایش داده می‌شود.

مقایسه	قسمت (و) (منهتن)	قسمت (ه) (منهتن + تضادهای خطی)	نتیجه
پیچیدگی زمانی	بررسی تعداد بیشتری از گره‌ها	بررسی تعداد کمتری از گره‌ها	**قسمت (ه) سریع‌تر است**
پیچیدگی فضایی	حافظه بیشتری نیاز دارد	حافظه کمتری نیاز دارد	**قسمت (ه) بهتر است**
معیار تعداد گره‌های Q	معیار تقریبی برای زمان است	همچنان معیار تقریبی است	**تعداد گره‌های Q معیار خوبی است، اما کافی نیست**