CP468 Assignment 1

A*, 8-Puzzle, 15-Puzzle, 24-Puzzle

Group 8

October 26, 2024

Team Members:

Jenish Bharucha, Romin Gandhi, Nakul Patel, Arsh Patel, Dhairya Patel
Paarth Bagga, Devarth Trivedi, Gleb Silin, Emmet Currie, Parker Riches

1 Code Implementation

Below is the Python code implementation for solving classic puzzles, including the 8-Puzzle, 15-Puzzle, and 24-Puzzle, using the A* search algorithm. This implementation applies various heuristic functions—such as misplaced tiles, Manhattan distance, and linear conflict—to evaluate and optimize moves towards the goal state efficiently.

2 8 Puzzle Code

```
import heapq
import random
import copy
import pandas as pd

GOAL_STATE = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]  # 0 represents the empty tile

#Helper function to print a puzzle state.
def print_puzzle(puzzle):
    for row in puzzle:
        print(row)
    print()

#Find the position of a tile in the puzzle.
def find_position(puzzle, value):
    for i, row in enumerate(puzzle):
        if value in row:
            return i, row.index(value)

#Heuristic h1: Misplaced tiles.
def misplaced_tiles(puzzle):
    return sum(1 for i in range(3) for j in range(3) if puzzle[i][j] != 0 and puzzle[i][j] != GOAL_STATE[i][j])

#Heuristic h2: Manhattan distance.
def manhattan_distance(puzzle):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = puzzle[i][j]
            if value != 0:
                x_goal, y_goal = find_position(GOAL_STATE, value)
                distance += abs(i - x_goal) + abs(j - y_goal)
    return distance

#Heuristic h3: Linear conflict.
def linear_conflict(puzzle):
    linear_conflict = manhattan_distance(puzzle)
    for i in range(3):
        for j in range(3):
            tile = puzzle[i][j]
            if tile != 0 and find_position(GOAL_STATE, tile)[0] == i:
```

```python
                for k in range(j + 1, 3):
                    if puzzle[i][k] != 0 and find_position(GOAL_STATE, puzzle[i][k])[0] == i and puzzle[i][k] < tile:
                        linear_conflict += 2
            if tile != 0 and find_position(GOAL_STATE, tile)[1] == j:
                for k in range(i + 1, 3):
                    if puzzle[k][j] != 0 and find_position(GOAL_STATE, puzzle[k][j])[1] == j and puzzle[k][j] < tile:
                        linear_conflict += 2
    return linear_conflict

#Generate n unique solvable puzzles.
def generate_reachable_puzzles(n=100):
    puzzles = set()
    print("Generating 100 unique, reachable puzzles...")
    while len(puzzles) < n:
        puzzle = generate_random_puzzle()
        puzzles.add(tuple(map(tuple, puzzle)))
    return [list(map(list, puzzle)) for puzzle in puzzles]

#Generate a random solvable puzzle state.
def generate_random_puzzle():
    puzzle = [1, 2, 3, 4, 5, 6, 7, 8, 0]
    while True:
        random.shuffle(puzzle)
        state = [puzzle[i:i + 3] for i in range(0, 9, 3)]
        if is_solvable(state):
            return state

#Check if the puzzle state is solvable.
def is_solvable(puzzle):
    flat_puzzle = [tile for row in puzzle for tile in row if tile != 0]
    inversions = sum(1 for i in range(len(flat_puzzle)) for j in range(i + 1, len(flat_puzzle)) if flat_puzzle[i] > flat_puzzle[j])
    return inversions % 2 == 0

#Perform A* search on a given puzzle using the specified heuristic.
def a_star(puzzle, heuristic):
    def get_neighbors(puzzle):
        neighbors = []
        x, y = find_position(puzzle, 0)
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
        for dx, dy in moves:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_puzzle = copy.deepcopy(puzzle)
                new_puzzle[x][y], new_puzzle[nx][ny] = new_puzzle[nx][ny], new_puzzle[x][y]
                neighbors.append(new_puzzle)
        return neighbors

```

```
    open_list = []
    heapq.heappush(open_list, (0 + heuristic(puzzle), 0, puzzle, None))
    visited = set()
    steps = 0
    nodes_expanded = 0

    while open_list:
        _, g, current, _ = heapq.heappop(open_list)
        steps += 1

        if current == GOAL_STATE:
            return g, nodes_expanded

        visited.add(tuple(map(tuple, current)))

        for neighbor in get_neighbors(current):
            if tuple(map(tuple, neighbor)) not in visited:
                nodes_expanded += 1
                heapq.heappush(open_list, (g + 1 + heuristic(neighbor), g + 1, neighbor, current))

    return -1, nodes_expanded

if __name__ == "__main__":
    puzzles = generate_reachable_puzzles(100)
    results = []  # Store all results

    for i, puzzle in enumerate(puzzles):
        print(f"Initial State for Puzzle {i + 1}:")
        print_puzzle(puzzle)

        h1_steps, h1_nodes = a_star(puzzle, misplaced_tiles)
        h2_steps, h2_nodes = a_star(puzzle, manhattan_distance)
        h3_steps, h3_nodes = a_star(puzzle, linear_conflict)

        print(f"Goal state achieved for Puzzle {i + 1}!\n")

        results.append({
            'Puzzle': i + 1,
            'Initial State': str(puzzle),
            'Steps (h1)': h1_steps, 'Nodes (h1)': h1_nodes,
            'Steps (h2)': h2_steps, 'Nodes (h2)': h2_nodes,
            'Steps (h3)': h3_steps, 'Nodes (h3)': h3_nodes
        })

    # Ensure full table display without truncation
    pd.set_option('display.max_rows', None)  # Show all rows
    pd.set_option('display.max_colwidth', None)  # Prevent column truncation

    df = pd.DataFrame(results)
    print(df)
```

2.1 Performance Analysis

From the data displayed above, various conclusions can be drawn on the performance of each of the 3 heuristics: **h1** (Misplaced Tile), **h2** (Manhattan Distance), and **h3** (Linear Conflict). A quick glance at the 3 tables shows that **h3** outperforms both **h2** and **h1**, and **h2** outperforms **h1**. Here are the details of each heuristic:

h1: This is the least optimal search compared to the other 2 heuristics, as it doesn't factor in the distance from the tile to the goal and only counts how many tiles are misplaced. This leads to very suboptimal performance, causing more nodes to be expanded in order to find the solution, making it inefficient compared to the other heuristics.

h2: This heuristic is more optimal than **h1**. The reason for this is that this heuristic factors in the distance from each tile to its goal position, leading to a more accurate heuristic than just counting misplaced tiles, expanding fewer nodes than **h1**, making it more efficient and optimal.

h3: This heuristic is the most optimal of the 3. The reason for this is that it utilizes the Manhattan Distance as well as applying a penalty for tiles that are in the correct column or row but simply out of order. It is the most informed heuristic out of the 3 and finds the solution path with the least number of nodes, making it the most efficient and optimal. The heuristics ordered in ranking of how informed they are is as follows: **h3** > **h2** > **h1**. This ranking also accurately depicts the performance level of each heuristic, where fewer nodes explored equates to a more efficient heuristic and faster achievement of the goal.

3 15 Puzzle Code

```
import random

class Node:
    def __init__(self, data, level, fvalue):
        # Initialize node with a matrix (data), level (depth), and
            f-value (A* evaluation)
        self.data = data
        self.level = level
        self.fvalue = fvalue

    # Locate the position of the blank space ('_')
    def locate(self, puzzle, x):
        for i in range(0, len(self.data)):
            for k in range(0, len(self.data)):
                if puzzle[i][k] == x:
                    return i, k

```

```python
    # Move the blank space with an adjacent tile, creating a new puzzle state
    def shuffle(self, puzzle, x1, y1, x2, y2):
        if 0 <= x2 < len(self.data) and 0 <= y2 < len(self.data[0]):
            temp_puzzle = self.copy(puzzle)  # Create a copy of the puzzle
            temp_puzzle[x2][y2], temp_puzzle[x1][y1] = temp_puzzle[x1][y1], temp_puzzle[x2][y2]  # Swap
            return temp_puzzle
        return None

    # Create subnodes by moving the blank space in four directions (left, right, up, down)
    def subnode(self):
        # Locate the current position of the blank space ('_')
        x, y = self.locate(self.data, '_')

        # Define potential moves for the blank space: left, right, up, down
        positions = [
            [x, y - 1],  # Move left
            [x, y + 1],  # Move right
            [x - 1, y],  # Move up
            [x + 1, y]   # Move down
        ]

        subnodes = []
        # Iterate over each potential move and create a subnode if the move is valid
        for new_x, new_y in positions:
            # Ensure that the new position is within the boundaries of the matrix (0 to size-1)
            if 0 <= new_x < len(self.data) and 0 <= new_y < len(self.data[0]):
                # Attempt to shuffle the blank space and create a new subnode
                node1 = self.shuffle(self.data, x, y, new_x, new_y)
                if node1 is not None:
                    # Create a new Node with the updated puzzle state
                    node2 = Node(node1, self.level + 1, 0)
                    subnodes.append(node2)

        return subnodes  # Return the list of generated subnodes


    # Create a deep copy of the puzzle matrix
    def copy(self, root):
        return [list(row) for row in root]


class Puzzle:
    def __init__(self, size):
        self.n = size  # Size of the puzzle grid (e.g., 4 for a 4x4 grid)
```

```python
        self.heuristic = ''  # Choose heuristic: 'h1' for Misplaced Tiles, 'h2' for Manhattan Distance
        self.open = []
        self.closed = []
        self.nodes_expanded = 0  # Counter for nodes expanded
        self.steps_taken = 0 #counter for steps taken

    def action(self, heuristic, start, goal):
        self.heuristic = heuristic
        start = Node(start, 0, 0)  # Create a Node for the initial state
        start.fvalue = self.f(start, goal)  # Calculate the f-value for the start node
        self.open.append(start)  # Add the start node to the open list
        self.nodes_expanded = 0  # Reset nodes expanded counter
        self.steps_taken = 0 # reset steps taken counter


        while True:
            current = self.open[0]  # Get the first node in the open list (node with smallest f-value)

            if self.h(current.data, goal) == 0:  # If the current state matches the goal state
                self.steps_taken = current.level
                break

            # Generate and evaluate subnodes (neighboring states)
            for i in current.subnode():
                i.fvalue = self.f(i, goal)  # Calculate the f-value for each subnode
                self.open.append(i)
                self.nodes_expanded += 1  # Increment nodes expanded counter

            self.closed.append(current)  # Add the current node to the closed list
            del self.open[0]  # Remove the current node from the open list
            self.open.sort(key=lambda x: x.fvalue)  # Sort the open list by f-value (ascending order)

    def f(self, start, goal):
        # Calculate the f-value: f(x) = g(x) + h(x)
        return self.h(start.data, goal) + start.level

    def h(self, start, goal):
        # Choose the appropriate heuristic
        if self.heuristic == 'h1':
            return self.heuristic_misplaced_tiles(start, goal)
        elif self.heuristic == 'h2':
            return self.heuristic_manhattan_distance(start, goal)
        elif self.heuristic == 'h3':
            return self.heuristic_linear_conflict(start, goal)
        else:
            raise ValueError("Invalid heuristic selected")
```

```python

    """ Heuristic h1: Counts the number of misplaced tiles """
    def heuristic_misplaced_tiles(self, start, goal):
        misplaced = 0
        for i in range(self.n):
            for j in range(self.n):
                if start[i][j] != goal[i][j] and start[i][j] != "_":
                    misplaced += 1
        return misplaced

    """ Heuristic h2: Calculates the Manhattan distance """
    def heuristic_manhattan_distance(self, start, goal):
        distance = 0
        goal_positions = {goal[i][j]: (i, j) for i in range(self.n) for j in range(self.n)}
        for i in range(self.n):
            for j in range(self.n):
                if start[i][j] != "_" and start[i][j] in goal_positions:
                    x_goal, y_goal = goal_positions[start[i][j]]
                    distance += abs(i - x_goal) + abs(j - y_goal)
        return distance

    """Heuristic h3: Linear Conflict, combining Manhattan Distance with linear conflicts"""
    def heuristic_linear_conflict(self, start, goal):
        manhattan_distance = self.heuristic_manhattan_distance(start, goal)
        linear_conflict = 0

        for row in range(self.n):
            row_conflict = self.find_conflicts(start[row], goal[row])
            linear_conflict += row_conflict

        for col in range(self.n):
            col_start = [start[row][col] for row in range(self.n)]
            col_goal = [goal[row][col] for row in range(self.n)]
            col_conflict = self.find_conflicts(col_start, col_goal)
            linear_conflict += col_conflict

        return manhattan_distance + 2 * linear_conflict

    """Identify conflicts in a row or column between start and goal states"""
    def find_conflicts(self, line_start, line_goal):
        conflict_count = 0
        for i in range(len(line_start)):
            for j in range(i + 1, len(line_start)):
                if (
                    line_start[i] != '_' and line_start[j] != '_'
                    and line_start[i] in line_goal and line_start[j] in line_goal
                    and line_goal.index(line_start[i]) > line_goal.index(line_start[j])
                ):
```

```
                    conflict_count += 1
        return conflict_count

    """Generate a random, reachable 15-puzzle state."""
    def generate_random_state(self, moves):
        goal_state = [['1', '2', '3', '4'],
                      ['5', '6', '7', '8'],
                      ['9', '10', '11', '12'],
                      ['13', '14', '15', '_']]

        # Copy the goal state to avoid modifying the original
        state = [row[:] for row in goal_state]
        x, y = 3, 3  # Position of the blank space in the goal state

        # Define the possible moves: left, right, up, down
        moves_list = [(-1, 0), (1, 0), (0, -1), (0, 1)]

        for _ in range(moves):
            # Select a random move
            move = random.choice(moves_list)
            new_x, new_y = x + move[0], y + move[1]

            # Ensure the move is within the puzzle boundaries
            if 0 <= new_x < self.n and 0 <= new_y < self.n:
                # Swap the blank space with the adjacent tile
                state[x][y], state[new_x][new_y] = state[new_x][new_y], state[x][y]
                x, y = new_x, new_y

        return state

    def print_puzzle(self, state):
        """Utility function to print the current puzzle state."""
        for row in state:
            print(" ".join(row))
        print()

results = []

# 4x4 matrix for 15-puzzle with heuristic selection
for i in range(100):
    start_state = Puzzle(4).generate_random_state(moves=30)
    goal_state = [['1', '2', '3', '4'],
                  ['5', '6', '7', '8'],
                  ['9', '10', '11', '12'],
                  ['13', '14', '15', '_']]  # Fixed goal state

    puzzle_h1 = Puzzle(4)
    puzzle_h1.action(heuristic='h1', start=start_state, goal=goal_state)

    # Store results for h1
    result_h1 = {
        "test_case": i + 1,
        "heuristic": 'h1',
        "steps_taken": puzzle_h1.steps_taken,
```

```
        "nodes_expanded": puzzle_h1.nodes_expanded
    }

    puzzle_h2 = Puzzle(4)
    puzzle_h2.action(heuristic='h2', start=start_state, goal=goal_state)

    # Store results for h2
    result_h2 = {
        "test_case": i + 1,
        "heuristic": 'h2',
        "steps_taken": puzzle_h2.steps_taken,
        "nodes_expanded": puzzle_h2.nodes_expanded
    }

    puzzle_h3 = Puzzle(4)
    puzzle_h3.action(heuristic='h3', start=start_state, goal=goal_state)

    # Store results for h3
    result_h3 = {
        "test_case": i + 1,
        "heuristic": 'h3',
        "steps_taken": puzzle_h3.steps_taken,
        "nodes_expanded": puzzle_h3.nodes_expanded
    }

    results.append({
        "test_case": i + 1,
        "h1_steps_taken": result_h1['steps_taken'],
        "h1_nodes_expanded": result_h1['nodes_expanded'],
        "h2_steps_taken": result_h2['steps_taken'],
        "h2_nodes_expanded": result_h2['nodes_expanded'],
        "h3_steps_taken": result_h3['steps_taken'],
        "h3_nodes_expanded": result_h3['nodes_expanded']
    })

    print("Test Case:", i + 1)
    for i in start_state:  # Print the current puzzle state
        for k in i:
            print(k, end=" ")
        print("")
    print("\n")



print("\nResults:")
print(f"{'Reachable':<10} | {'h1 Steps':<10} | {'h1 Nodes':<12} | {'h2 Steps':<10} | {'h2 Nodes':<12} | {'h3 Steps':<10} | {'h3 Nodes':<12}")
print("-" * 95)
for result in results:
    print(f"{result['test_case']:<10} | {result['h1_steps_taken']:<10} | {result['h1_nodes_expanded']:<12} | {result['h2_steps_taken']:<10} | {result['h2_nodes_expanded']:<12} | {result['h3_steps_taken']:<11}| {result['h3_nodes_expanded']:<12}" )
```

3.1 Performance Analysis

Misplaced Tiles (h1):

- This heuristic is the least informed among the three, and it only counts tiles that are out of their goal positions.
- Since it doesn't consider the actual distance each tile needs to move, it tends to generate higher node expansions compared to **h2** and **h3**.
- While **h1** still guarantees a solution with the minimum number of moves due to the A* algorithm's properties, it typically requires significantly more node expansions, leading to slower and less efficient searches.

Manhattan Distance (h2):

- **h2**, which computes the total Manhattan distance of tiles from their goal positions, is more informed than **h1**.
- Since it reflects both the position and distance of each tile to its target, **h2** results in more selective and effective expansions.
- In most cases, **h2** explores fewer nodes than **h1**, and its paths are generally optimal. This makes it a more efficient choice over **h1** for the 15 Puzzle.

Linear Conflict (h3):

- **h3** combines **h2** with an analysis of linear conflicts (where two tiles in the same row or column block each other), making it the most informed heuristic.
- By adding a penalty for these conflicts, **h3** reduces the need for additional expansions in such cases, further refining the efficiency of node expansions.
- As a result, **h3** generally outperforms **h1** and **h2** in cases with significant linear conflicts, exploring the fewest nodes among the three while maintaining the minimum solution path length.

4 24 Puzzle Code

```
import heapq
import random
import copy
import pandas as pd

# Define the 5x5 goal state for the 24-puzzle
GOAL_STATE = [[1, 2, 3, 4, 5],
              [6, 7, 8, 9, 10],
              [11, 12, 13, 14, 15],
              [16, 17, 18, 19, 20],
```

```python
                [21, 22, 23, 24, 0]]  # 0 is the blank tile

#Helper function to print a puzzle state."""
def print_puzzle(puzzle):
    for row in puzzle:
        print(row)
    print()

#Find the position of a tile in the puzzle.
def find_position(puzzle, value):
    for i, row in enumerate(puzzle):
        if value in row:
            return i, row.index(value)

#Heuristic h1: Misplaced tiles.
def misplaced_tiles(puzzle):
    return sum(1 for i in range(5) for j in range(5) if puzzle[i][j] != 0 and puzzle[i][j] != GOAL_STATE[i][j])

#Heuristic h2: Manhattan distance.
def manhattan_distance(puzzle):
    distance = 0
    for i in range(5):
        for j in range(5):
            value = puzzle[i][j]
            if value != 0:
                x_goal, y_goal = find_position(GOAL_STATE, value)
                distance += abs(i - x_goal) + abs(j - y_goal)
    return distance

#Heuristic h3: Linear conflict.
def linear_conflict(puzzle):
    linear_conflict = manhattan_distance(puzzle)
    for i in range(5):
        for j in range(5):
            tile = puzzle[i][j]
            if tile != 0 and find_position(GOAL_STATE, tile)[0] == i:
                for k in range(j + 1, 5):
                    if puzzle[i][k] != 0 and find_position(GOAL_STATE, puzzle[i][k])[0] == i and puzzle[i][k] < tile:
                        linear_conflict += 2
            if tile != 0 and find_position(GOAL_STATE, tile)[1] == j:
                for k in range(i + 1, 5):
                    if puzzle[k][j] != 0 and find_position(GOAL_STATE, puzzle[k][j])[1] == j and puzzle[k][j] < tile:
                        linear_conflict += 2
    return linear_conflict

#Generate n unique solvable 24-puzzles.
def generate_reachable_puzzles(n=100):
    puzzles = set()
    print("Generating 100 unique, reachable puzzles...")
    while len(puzzles) < n:
```

```
        puzzle = generate_random_puzzle()
        puzzles.add(tuple(map(tuple, puzzle)))
    return [list(map(list, puzzle)) for puzzle in puzzles]

#Generate a random solvable puzzle state.
def generate_random_puzzle():
    puzzle = list(range(1, 25)) + [0]  # Numbers 1-24 with a blank (0)
    while True:
        random.shuffle(puzzle)
        state = [puzzle[i:i + 5] for i in range(0, 25, 5)]
        if is_solvable(state):
            return state

#Check if the puzzle state is solvable.
def is_solvable(puzzle):
    flat_puzzle = [tile for row in puzzle for tile in row if tile != 0]
    inversions = sum(1 for i in range(len(flat_puzzle)) for j in range(i + 1, len(flat_puzzle)) if flat_puzzle[i] > flat_puzzle[j])
    blank_row = next(i for i, row in enumerate(puzzle) if 0 in row)
    return (inversions + blank_row) % 2 == 0

#Perform A* search on a given puzzle using the specified heuristic.
def a_star(puzzle, heuristic):
    def get_neighbors(puzzle):
        neighbors = []
        x, y = find_position(puzzle, 0)
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
        for dx, dy in moves:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 5 and 0 <= ny < 5:
                new_puzzle = copy.deepcopy(puzzle)
                new_puzzle[x][y], new_puzzle[nx][ny] = new_puzzle[nx][ny], new_puzzle[x][y]
                neighbors.append(new_puzzle)
        return neighbors

    open_list = []
    heapq.heappush(open_list, (0 + heuristic(puzzle), 0, puzzle, None))
    visited = set()
    steps = 0
    nodes_expanded = 0

    while open_list:
        _, g, current, _ = heapq.heappop(open_list)
        steps += 1

        if current == GOAL_STATE:
            return g, nodes_expanded

        visited.add(tuple(map(tuple, current)))

        for neighbor in get_neighbors(current):
```

```python
                if tuple(map(tuple, neighbor)) not in visited:
                    nodes_expanded += 1
                    heapq.heappush(open_list, (g + 1 + heuristic(neighbor), g + 1, neighbor, current))

    return -1, nodes_expanded

if __name__ == "__main__":
    puzzles = generate_reachable_puzzles(100)
    results = []  # Store all results

    for i, puzzle in enumerate(puzzles):
        print(f"\nInitial State for Puzzle {i + 1}:\n")
        print_puzzle(puzzle)

        h1_steps, h1_nodes = a_star(puzzle, misplaced_tiles)
        h2_steps, h2_nodes = a_star(puzzle, manhattan_distance)
        h3_steps, h3_nodes = a_star(puzzle, linear_conflict)

        print(f"Goal state achieved for Puzzle {i + 1}!\n")

        results.append({
            'Puzzle': i + 1,
            'Initial State': str(puzzle),
            'Steps (h1)': h1_steps, 'Nodes (h1)': h1_nodes,
            'Steps (h2)': h2_steps, 'Nodes (h2)': h2_nodes,
            'Steps (h3)': h3_steps, 'Nodes (h3)': h3_nodes
        })

    pd.set_option('display.max_rows', None)  # Show all rows
    pd.set_option('display.max_colwidth', None)  # Prevent column truncation

    df = pd.DataFrame(results)
    print(df)
```

4.1 Performance Analysis

Heuristic **h1**, which counts tiles that are misplaced, typically performs the least well. It requires more nodes to be expanded and more steps to find solutions. This is because it only gives a rough estimate of how close the goal is and not enough to effectively guide the search.

The efficiency of the Manhattan distance heuristic (**h2**) is significantly higher than that of **h1**. On average, it requires fewer steps and nodes to be expanded. It improves the efficiency of the **A* algorithm** by giving a more direct indicator of the distance to the goal state by taking into account the minimal number of movements needed for each tile.

The Manhattan distance is improved by Heuristic (**h3**), the linear conflict heuristic, which adds penalties for tiles in opposing locations. This improves search performance by promoting the settlement of local conflicts. Because of this, **h3** often has the fewest node expansions and steps, proving its advantage in directing the **A* search** in fewer movements toward the best answers.