جامعة الأخوين

ⵜⴰⵙⴷⴰⵡⵉⵜ ⵏ ⵓⵅⴰⵡⴰⵢⵏ

AL AKHAWAYN UNIVERSITY

SPRING 2024

CSC 4301 Artificial Intelligence

Project 1

- Report -

February 6th, 2024

Realized by: Chahd Maatallaoui <100676>, Imane Bouchtaoui <109903>

Supervised by: Dr. RACHIDI Tajjeddine

Abstract:

The implementation and empirical assessment of various heuristics created for the 8puzzle game—a persistent issue in the realm of artificial intelligence—are the main subjects of this work. In the 8puzzle game, tiles with numbers 1 through 8 and an empty space are rearranged inside a 3x3 grid. The main goal is to find out how well various heuristic approaches work for solving the 8puzzle game, with a focus on performance and optimization. This research aims to offer significant insights into the relative efficacy of these heuristics by methodical use and extensive testing, which aims to improve heuristic-based strategies for handling the difficulties presented by the 8puzzle game.

I. Introduction:

Before delving into the heuristic, it is essential to delineate and clarify the key components of our search problem project:

- **State Space:** This encompasses the puzzle's configuration, including the start state, goal test, and all successor functions leading to a solution.
- **Initial Space:** Symbolizing the puzzle's initial configuration, a 3x3 grid with eight numbers in constant shuffling alongside a blank tile. The initial state varies across problem instances due to randomized positions.
- **Goal State:** This represents the optimal state where agents cease transitioning. The tiles should be arranged in ascending order, with the empty tile positioned in the top-left corner of the grid.
- **Successor Function:** These are the transitional states demonstrating the link between the start state and the goal test. Represented as an order adjusting the blank tile toward the goal test, using the verbs "Move" and designated directions "Up, Down, Left, Right."
- **Solution:** This denotes a path or distance indicating various successive states leading to the goal test, revealing the employed strategy.

- **World State:** Encompasses all possible combinations of the entire search problem.

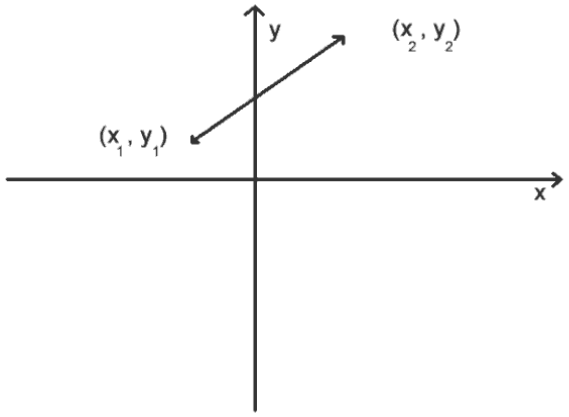
A central focus of this research involves introducing four heuristic functions integral to informed search algorithms like "A* Search." These heuristic functions provide cost estimates required to find a solution for the Eight Puzzle. The research will introduce and analyse four heuristics, outlining their implementation and impact within the context of the Eight Puzzle.

H1/ Misplaced Tiles Count: This heuristic quantifies the number of tiles out of position at the puzzle's conclusion, providing an estimate of the state's distance from the solution. Correctly positioned tiles are excluded from the count.

H2/ Sum of Euclidean Distances: Using the Euclidean distance between each tile's current and goal positions, this heuristic offers a geometrically informed estimate of the state's distance to the solution. The Euclidean distance formula measures the straight-line distance between two points in a two-dimensional space. For a tile problem where each tile is represented by coordinates (x,y), and the goal position is ($goal_x$,$goal_y$), the Euclidean distance is calculated as follows:

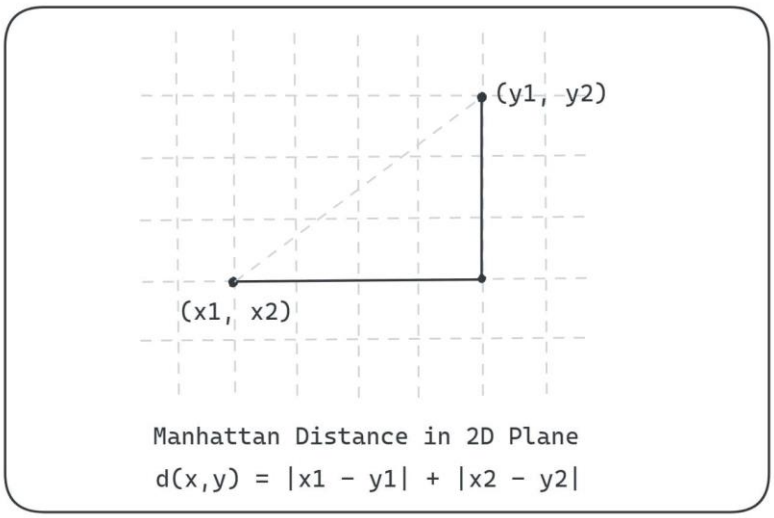
$$D\text{Euclidean}=\sqrt{(x-\text{goal}_x)^2+(y-\text{goal}_y)^2}$$

$$d=\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$



H3/ Sum of Manhattan Distances: Calculating the sum of horizontal and vertical distances for each tile to reach its goal, the Manhattan distance provides an estimate of the number of moves required and reveals the misalignment of the entire grid. The Manhattan distance measures the distance between two points on a grid using the sum of the absolute differences in their x and y coordinates. For a current tile at coordinates (x, y) and its goal position at ($goal_x$, $goal_y$), the Manhattan Distance is calculated as:

$$D_{\text{Manhattan}} = | x - \text{goal}_x | + | y - \text{goal}_y |$$



H4/ Number of Tiles Out of Row and Column (Row and Column Deviation): Evaluating the deviation of tiles within each row and column from their proper positions, this heuristic considers both row-wise and column-wise aspects, providing a unique estimation of the state evaluation.

In this study, we will explore and analyze these heuristics. Our approach involves comprehending the implementation of each heuristic and conducting a thorough comparison to discern their respective efficiency and results' optimality in solving the Eight Puzzle problem.

II. TASK 1: Implementations of the heuristics

- **Heuristic 1: number of misplaced tiles**

```python
def h1(state, problem = None):
    """
    - state: The current state of the puzzle.
    - out_of_place: The number of tiles that are misplaced.
    """

    out_of_place = 0
    goal_state = [[0,1,2], [3,4,5], [6,7,8]] #the goal state we would like to reach

    #we want to iterate through each cell in the puzzle
    for row in range(3):
        for col in range(3):
            #we will check if the value of the current cell is the same as the value of the corresponding cell in the goal state
            if state.cells[row][col] != goal_state[row][col]:
                out_of_place += 1

    #return the number of tiles that are out of place
    return out_of_place
```

The goal state here is a 3x3 grid where the tiles are organized in ascending order from 0 to 8 with 0 representing the empty space. It will therefore iterate through each tile of the current state of the puzzle and compare it to its expected position in the goal state. If it is different, it will increment the out-of-place variable (not including the empty space), which will be returned at the end.

- **Heuristic 2: Sum of Euclidian distance**

```
#heuristic for eucleadian sum from tile to goal
def h2(state, problem = None):
    """
    - state: The current state of the puzzle.
    - eucleadian_sum : the sum of all the eucleandian distances between the placement of the tile and where it is in the goal state
    ((x2-x1)**2 + (y2-y1)**2)**0.5
    """
    eucleadian_sum = 0 #variable where we will be storing the result and return it at the end
    grid = state.cells

    #iterate over each cell of the puzzle
    for row in range(3):
        for col in range(3):
            tile = grid[row][col]
            if tile != 0: #the tile that contains a blank space is ignored
                eucleandian_distance = 0 #variable created in order to store the eucleadian distance in each iteration
                goal_row = tile // 3  #the goal row is calculated by taking the value in the cell and dividing it by 3. the returned result
                goal_column = tile % 3 #the goal column is calculated by taking the modulo of the value of the cell by 3
                eucleadian_distance = ((row - goal_row) ** 2 + (col-goal_column)**2)**.5
                eucleadian_sum += eucleadian_distance #add the distance found to the one found in the previous iterations

    return eucleadian_sum
```

We calculate the goal row and column based on the value of the tile. Then, for each tile (aside from the empty space one), we calculate the Euclidean distance between its current position (row and column) and the goal position that has been calculated (goal_row and goal_column) using the formula. The distances of all tiles are then summed into one variable, eucleadian_sum, which is then returned.

- **Heuristic 3 : Sum of Manhattan distances**

```
#heuristic for manhattan sum from tile to goal
def h3(state, problem = None):
    """
    - state: The current state of the puzzle.
    - manhattan_sum : the sum of all the manhattan distances between the placement of the tile and where it is in the goal state
    abs(x2 - x1) + abs(y2 - y1)
    """
    #iterate through the puzzle
    manhattan_sum = 0 #variable where we will be storing the result and return it at the end
    grid = state.cells
    for row in range(3):
        for col in range(3):
            tile = grid[row][col]
            if(tile != 0): #we ignore the tile that is an empty space
                manhattan_distance = 0 #variable used to store the manhnattan distance in each iteration
                goal_row = tile // 3 #the goal row is calculated by taking the value in the cell and dividing it by 3. the returned result should be an integer
                goal_column = tile % 3 #the goal column is calculated by taking the modulo of the value of the cell by 3
                manhattan_distance = abs(row-goal_row) + abs(col-goal_column)
                manhattan_sum += manhattan_distance

    return manhattan_sum
```

Using the value of each tile, we calculate the goal_row and goal_column. The Manhattan distance is then calculated by computing the absolute difference between the current row and the goal_row + the difference between the current column and the goal_column. It gives us the minimum number of moves

required to get from the current position to the goal one by moving only vertically or horizontally. We then sum the Manhattan distances of all the tiles in one variable, manhattan_sum, which will be returned at the end.

- **Heuristic 4: Number of Tiles Out of Row and Column (Row and Column Deviation)**

```
# heuristic to find the number of tiles out of their correct row and column
def h4(state, problem = None):
    """
    - state: The current state of the puzzle.
    - total_misplaced : the sum of the tiles that are not in their goal row + the tiles that are not in their goal column
    abs(x2 - x1) + abs(y2 - y1)
    """
    total_misplaced = 0  # Initialize the total count of misplaced tiles
    for i in range(3):  # Iterate over each row
        for j in range(3):  # Iterate over each column
            current_value = state.cells[i][j]  # Get the value of the current cell
            if current_value != 0:  # Ignore the blank cell
                # Check if the current tile is misplaced in the row and column
                misplaced_row = current_value // 3 != i
                misplaced_column = current_value % 3 != j
                # Increment the count if the tile is misplaced either in row or column
                total_misplaced += int(misplaced_row) + int(misplaced_column)
    return total_misplaced  # Return the total count of misplaced tiles
```

We iterate through the puzzle and for each tile (ignoring the empty space), we check whether the tile is in its correct row and column using the value of the tile. If it is not the case, we increment either misplaced_row or misplaced_column or both. We then sum them up and add them to a total_misplaced variable which holds the misplaced positions of all the misplaced rows and columns of all the tiles, which will then be returned.

III. Task 2: Comparisons of the heuristics

Choosing a search method is essential to solving the 8-puzzle issue and getting the best results. This puzzle represents a wider range of combinatorial search issues in addition to being a classic. To reduce the number of movements needed to achieve the goal state, navigating its solution requires making strategic judgments across a complex state space. Different search algorithms tackle this problem in varying degrees of optimality and efficiency.

H1: based on the number of misplaced tiles, offers simplicity and intuitive insight. Its straightforward computation involves counting tiles in incorrect positions, providing a quick assessment of a state's proximity to the goal. This simplicity facilitates efficient evaluation. However, H1 lacks consideration for distances between misplaced and correct positions, potentially leading to suboptimal solutions. Its focus on minimizing misplaced tiles may overlook more intricate state configurations, impacting overall optimality.

H2: determined by summing the Euclidean distances of tiles from their goal positions, brings notable advantages and challenges to the 8-puzzle problem. Unlike simpler heuristics, H2 considers the actual spatial distances between tiles and their intended goal locations, offering a more accurate assessment of a state's arrangement. This nuanced approach enhances the heuristic's ability to guide the search algorithm towards more optimal solutions, favoring states with shorter paths to the goal. However, the inclusion of Euclidean distance calculations increases computational complexity, potentially affecting overall efficiency. Additionally, there is a risk of overestimation in certain configurations, leading to exploration of suboptimal paths.

H3: by considering horizontal and vertical movements, H3 efficiently gauges a state's proximity to the goal without the computational complexity of Euclidean distances. This approach improves optimality by favoring states with minimized Manhattan distances, contributing to more efficient paths. However, H3 overlooks diagonal movements, potentially impacting precision in certain scenarios. Additionally, there's a risk of overestimation, influencing the heuristic's ability to consistently guide the search towards the most efficient solutions. Despite these considerations, H3 remains a valuable and pragmatic heuristic, balancing accuracy and computational efficiency in solving the 8-puzzle problem.

H4: determined by the sum of tiles out of their correct rows and columns, offers a straightforward and practical approach to the 8-puzzle problem. Its simplicity involves a direct count of misaligned tiles, facilitating efficient computation for quick state assessments. By emphasizing alignment in both rows and columns, H4 provides a direct indication of a state's proximity to the goal configuration. However, H4's focus on row and column positions may lead to underestimating the true number of moves required, especially when intricate spatial relationships and precise movements are essential. Although the simplistic method is fast for alignment evaluations, it is not able to fully capture the complexities of the 8-puzzle problem, leading to less-than-ideal estimates for possible paths of solution.

- "Code to generate different scenarios in automate.py and saving them to a scenarios.csv file"

```
def get_scenarios(moves=3, n=200):
    scenarios = []
    for i in range(n):
        puzzle = eightpuzzle.createRandomEightPuzzle(moves)
        elements = []
        for row in puzzle.cells:
            for cell in row:
                elements.append(cell)
        scenarios.append(tuple(elements))
    return scenarios

def save_csv(scenarios, file_name):
    with open(file_name, 'w', newline='') as csvfile:
        writer = csv.writer(csvfile)
        for scenario in scenarios:
            writer.writerow(scenario)
    print(f"Scenarios saved to {file_name}.")
```

- Functions to automate the applications of the different heuristics to the scenarios:

```python
def load_scenarios(filename='scenarios2.csv'):
    scenarios = []
    with open(filename, 'r') as csvfile:
        reader = csv.reader(csvfile)
        for row in reader:
            scenario = list(map(int, row))
            scenarios.append(scenario)
    print(f"Loaded {len(scenarios)} scenarios from {filename}.")
    return scenarios
```

```python
def run_search(problem, search_method, heuristic=None):
    try:
        heuristic_name = heuristic.__name__ if heuristic else 'None'
        print(f"Running search: {search_method.__name__}, heuristic: {heuristic_name}")

        if heuristic:
            result = search_method(problem, heuristic)
        else:
            result = search_method(problem)

        if len(result) == 4:  # Assuming result format is (path, expanded_nodes, fringe_size, depth)
            return result
        else:
            raise ValueError("Incorrect return format")

    except Exception as e:
        print(f"Error during search with {search_method.__name__}: {e}")
        return [], 0, 0, 0  # Default return format in case of an error
```

```python
def compare_methods(scenarios, methods, output_file='results2.csv'):
    results = []
    print(f"Comparing search methods with {len(scenarios)} scenarios.")

    for index, scenario in enumerate(scenarios):
        state = eightpuzzle.EightPuzzleState(scenario)  # Adjust for your problem's state initialization
        problem = eightpuzzle.EightPuzzleSearchProblem(state)  # Adjust for your problem definition

        for method_name, method_details in methods.items():
            search_method = method_details['method']
            heuristic = method_details.get('heuristic')

            path, expanded_nodes, fringe_size, depth = run_search(problem, search_method, heuristic)

            # Convert the path to a string representation, if it's not already
            path_str = '->'.join(str(step) for step in path) if path else "No Path"

            results.append({
                'scenario': f"Scenario {index + 1}",
                'method': method_name,
                'expanded_nodes': expanded_nodes,
                'fringe_size': fringe_size,
                'depth': depth,
                'path': path_str  # Including path in the results
            })

    with open(output_file, 'w', newline='') as csvfile:
        fieldnames = ['scenario', 'method', 'expanded_nodes', 'fringe_size', 'depth', 'path']
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        for result in results:
            writer.writerow(result)

    print(f"Results written to {output_file}.")
```

```
def main():
    scenarios = get_scenarios(10, 100)
    save_csv(scenarios, 'scenarios2.csv')
    scenarios = load_scenarios()

    methods = {
        'BFS': {'method': search.breadthFirstSearch},
        'DFS': {'method': search.depthFirstSearch},
        'UCS': {'method': search.uniformCostSearch},
        'A*': {'method': search.aStarSearch, 'heuristic': search.nullHeuristic},
        'h3': {'method': search.aStarSearch, 'heuristic': search.h3}
    }
    compare_methods(scenarios, methods)


   # Load your CSV file
    df = pd.read_csv('results2.csv')

    # Filter out DFS results


    # Calculate the averages for each method
    average_metrics = df.groupby('method')[['expanded_nodes', 'fringe_size', 'depth']].mean()

    print(average_metrics)


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

->Please run the automate.py file to get results

After generating 200 different scenarios for the 8-puzzle and applying the different heuristics to each one of these scenarios and recording the expanded nodes, fringe size, depth and path, we get the following results (please refer to the scenarios1.csv and results1.csv files in the folder to get the results of all 200 scenarios.):

scenarios1.csv

```
4,3,0,1,5,2,6,7,8
1,2,5,3,4,8,0,6,7
3,1,0,4,5,2,6,7,8
1,4,2,3,0,5,6,7,8
0,2,5,1,3,4,6,7,8
1,4,2,3,0,5,6,7,8
3,2,0,4,1,5,6,7,8
1,2,5,3,4,8,0,6,7
1,2,5,3,4,8,6,7,0
0,3,2,4,1,5,6,7,8
3,1,2,6,4,5,7,8,0
1,2,5,3,0,8,6,4,7
1,2,5,3,0,4,6,7,8
1,2,5,6,3,8,0,4,7
3,5,1,4,0,2,6,7,8
3,1,2,6,0,5,7,4,8
0,1,2,3,7,5,4,6,8
1,4,2,6,0,5,7,3,8
1,2,5,6,3,4,0,7,8
3,1,2,6,0,5,7,4,8
3,2,5,4,0,1,6,7,8
3,1,2,4,5,8,0,6,7
0,1,2,3,4,5,6,7,8
1,4,2,3,0,5,6,7,8
1,4,2,3,5,8,6,7,0
1,4,2,3,7,5,6,8,0
```

results1.csv

```
scenario,heuristic,expanded_nodes,fringe_size,depth,path
s1,h1,23,21,8,down->left->up->left->down->right->up->left
s1,h2,13,14,8,down->left->up->left->down->right->up->left
s1,h3,13,14,8,down->left->up->left->down->right->up->left
s1,h4,13,14,8,down->left->up->left->down->right->up->left
s2,h1,6,5,6,right->right->up->up->left->left
s2,h2,6,5,6,right->right->up->up->left->left
s2,h3,6,5,6,right->right->up->up->left->left
s2,h4,6,5,6,right->right->up->up->left->left
s3,h1,4,6,4,down->left->left->up
s3,h2,4,6,4,down->left->left->up
s3,h3,4,6,4,down->left->left->up
s3,h4,4,6,4,down->left->left->up
s4,h1,2,5,2,up->left
s4,h2,2,5,2,up->left
s4,h3,2,5,2,up->left
s4,h4,2,5,2,up->left
s5,h1,7,8,6,down->right->right->up->left->left
s5,h2,6,7,6,down->right->right->up->left->left
s5,h3,6,7,6,down->right->right->up->left->left
s5,h4,6,7,6,down->right->right->up->left->left
s6,h1,2,5,2,up->left
s6,h2,2,5,2,up->left
s6,h3,2,5,2,up->left
s6,h4,2,5,2,up->left
s7,h1,5,6,4,left->down->left->up
```

We averaged the results into the following table:

```
import pandas as pd
           expanded_nodes  fringe_size  depth
heuristic
h1                  5.845        6.945   4.08
h2                  4.660        6.105   4.08
h3                  4.650        6.085   4.08
h4                  4.710        6.135   4.08
```

After assessing all the previously provided information weighing all potential benefits and drawbacks, it is evident that the third heuristic—the "Sum of Manhattan Distances" heuristic—is the most effective and appropriate one to use in the eight-puzzle situation. It presents the least average expanded nodes as well the the least average fringe size compared to the other heuristics. On the other hand, the first heuristic is the least optimal as it presents the highest average of expanded nodes and the highest average for the fringe size. On the other hand, the average depth is the same in all 4 heuristics.

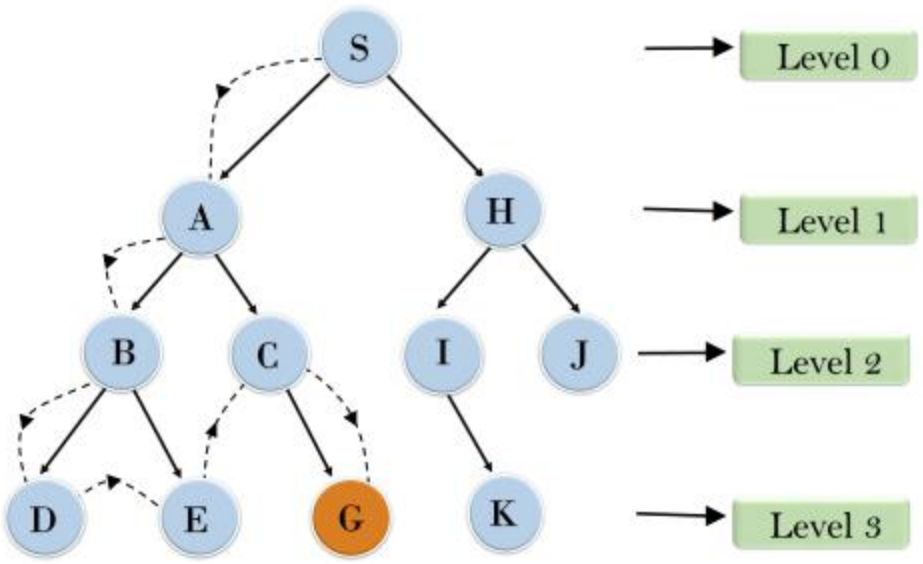
IV. Task 3: Overall Comparison between strategies

Upon recognizing the significance of heuristic adequacy, our next step was to implement the Manhattan algorithm along with various search strategies to identify the optimal combination ensuring the most efficient solution. We engaged with three uninformed and one informed search techniques to thoroughly evaluate and compare their performance in achieving optimal solutions.

"Uninformed Search"-> DFS, BFS, UCS traversals

DFS- Depth First Search (DFS) employs a Last in First Out (LIFO) strategy in its implementation. The process involves creating a stack to store states, initially pushing the starting state onto the stack. Subsequently, the algorithm iteratively pops states from the stack, checking if each is the goal state—a condition serving as the base case. If not, the algorithm generates all possible successor states from the current one, updating metrics such as fringe length and expanded nodes. Consequently, the depth and fringe metrics can unexpectedly inflate to large numbers. However, this method lacks optimality as it explores all nodes, potentially encountering loops that lead to infinite paths, resulting in a solution that is far from being optimal.

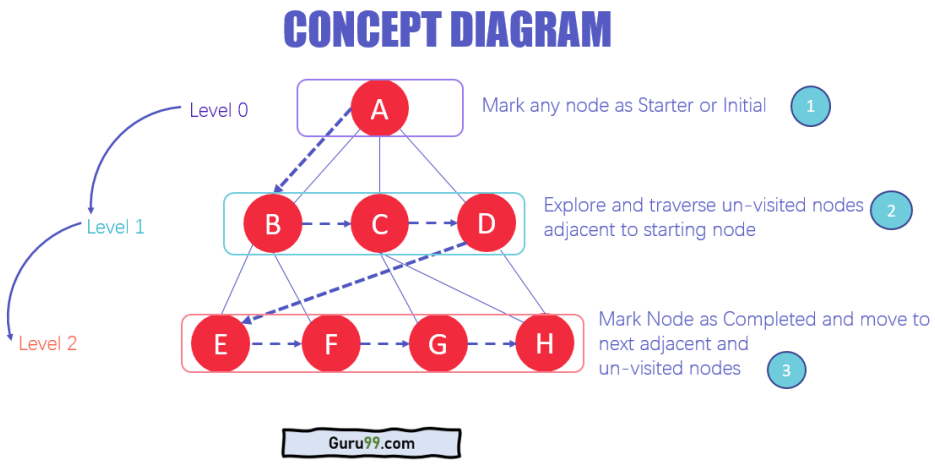
Depth First Search




```python
def depthFirstSearch(problem):
    # Initialize the fringe as a Stack for LIFO behavior.
    fringe = Stack()
    # Push the initial state, an empty action list, and initial depth.
    fringe.push((problem.getStartState(), [], 0))

    # Track visited states to avoid revisiting.
    exploredNodes = set()

    # Metrics for tracking the search process.
    expandedNodes = 0
    fringeSize = 0

    while not fringe.isEmpty():
        currentState, actions, depth = fringe.pop()  # Explore the next state.

        # Check goal state after popping, to ensure goal check even if state is revisited.
        if problem.isGoalState(currentState):
            return actions, expandedNodes, fringeSize, depth

        # Only proceed if we haven't explored this state yet.
        if currentState not in exploredNodes:
            exploredNodes.add(currentState)  # Mark the current state as explored.
            expandedNodes += 1

            # Limit the fringe to 500 elements.
            if fringe.size() >= 200:
                continue

            successors = problem.getSuccessors(currentState)
            for successor_State, successor_Action, _ in successors:
                # No need to check if successor_State is in exploredNodes here,
                # as we want to add all successors to the fringe for depth-first behavior.
                newActions = actions + [successor_Action]
                fringe.push((successor_State, newActions, depth + 1))

            fringeSize = max(fringeSize, fringe.size())  # Update the maximum fringe size observed.

    # Return defaults if goal not found.
    return [], expandedNodes, fringeSize, 0
```

We encountered this problem in our project as we were entered into a loop that kept running forever. The numbers generated would be too big and the code was unable to go to the next method. Therefore, for efficiency purposes, we set the limit of the fringe to 500.

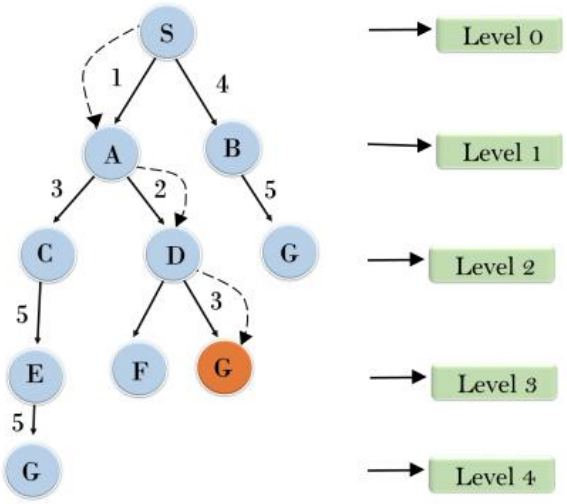
BFS- Breadth First Search (BFS) is based on the First in First Out (FIFO) strategy that involves creating a queue to store states and a list for tracking visited states. The process entails looping while the queue is not empty, dequeuing a state, and checking if it meets the goal test (base case). If not, the algorithm enqueues all possible successor states into the queue while adding the current state to the visited list. Despite its superior performance compared to DFS, BFS exhibits a significant time complexity, reaching exponential numbers, leading to prolonged solution times for generating successive moves. Additionally, BFS faces challenges with a larger fringe space compared to DFS, posing substantial memory usage concerns.



```python
def breadthFirstSearch(problem):
    fringe = Queue()
    exploredNodes = set()
    fringe.push((problem.getStartState(), []))
    expanded_nodes = 0
    currentFringeSize = 1  # Initialize current fringe size
    fringeSize = currentFringeSize  # Track max fringe size

    while not fringe.isEmpty():
        currentState, actions = fringe.pop()
        currentFringeSize -= 1  # Decrement for each node popped

        if currentState not in exploredNodes:
            exploredNodes.add(currentState)
            expanded_nodes += 1

            successors = problem.getSuccessors(currentState)
            for succState, succAction, _ in successors:
                if succState not in exploredNodes:
                    fringe.push((succState, actions + [succAction]))
                    currentFringeSize += 1  # Increment for each node pushed
                    fringeSize = max(fringeSize, currentFringeSize)  # Update max fringe size

            if problem.isGoalState(currentState):
                return actions, expanded_nodes, fringeSize, len(actions)

    return [], expanded_nodes, fringeSize, 0
```

UCS- The Uniform Cost Search (UCS) method entails initializing a priority queue called "frontier," where the initial state is stored in the start node. This is accompanied by an empty list of actions with a cost set to 0. Every iteration of the process involves removing the node with the lowest cost until the queue is empty. The node is added to the list if it hasn't been visited yet and its cost is less than the amount that was registered. After the algorithm reaches the desired state, it outputs the sequence of moves performed, the number of enlarged nodes, and the maximum fringe length. Because UCS incorporates cost concerns, it is more efficient than other uninformed search methods, even in the absence of heuristics.

Uniform Cost Search




```python
def uniformCostSearch(problem):

    # Initialize a priority queue for the fringe to manage nodes based on the total path cost.
    fringe = util.PriorityQueue()
    # Dictionary to track the lowest cost at which we've reached each state.
    exploredNodes = {}
    startNode = (problem.getStartState(), [], 0)
    fringe.push(startNode, 0)
    # Variables to keep track of the number of nodes expanded and the maximum size of the fringe.
    expanded_nodes = 0
    fringeSize = 0

    while not fringe.isEmpty():
        fringeSize = max(fringeSize, fringe.size())

        # Pop the node with the lowest total path cost from the fringe.
        currentState, actions, currentCost = fringe.pop()

        if currentState not in exploredNodes or currentCost < exploredNodes[currentState]:
            # Record this state and its path cost as the best path found so far.
            exploredNodes[currentState] = currentCost
            expanded_nodes += 1  # Increment the count of expanded nodes.

            if problem.isGoalState(currentState):
                # If so, return the path found, number of nodes expanded, maximum fringe size, and path length.
                return actions, expanded_nodes, fringeSize, len(actions)

            # Generate successors of the current state and process each.
            for succState, succAction, succCost in problem.getSuccessors(currentState):
                newCost = currentCost + succCost  # Calculate the new total cost for reaching the successor.
                newNode = (succState, actions + [succAction], newCost)

                # Add the successor to the fringe if it hasn't been explored or if this is a cheaper path.
                if succState not in exploredNodes or newCost < exploredNodes.get(succState, float('inf')):
                    fringe.update(newNode, newCost)

    return [], expanded_nodes, fringeSize, 0
```

“Informed Search”-> A* traversal

A*- The A* algorithm prioritizes finding the most optimal solution. Like UCS, it initializes a priority queue, but this time the priority is rather determined by the sum of the path cost and a heuristic estimate, combining past results and future expectations through functions g(n) and h(n). While resembling UCS, the inclusion of heuristics, specifically the Manhattan distance, significantly distinguishes A* Search, making it more flexible than other techniques. A* Search, particularly with the *Manhattan* heuristic, strikes a balance between informed exploration, efficiency, completeness, optimality, and low space complexity. It fulfills the need for an informed search technique with an effective and admissible heuristic, making it indeed the ideal choice for solving the 8-puzzle problem.

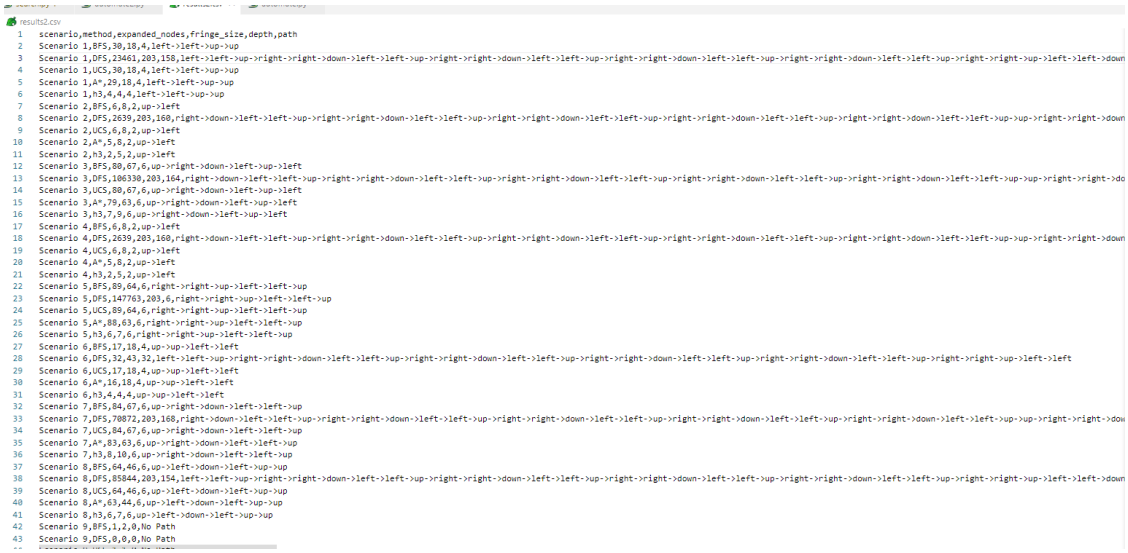
->Please run the automate2.py file to get results

- Functions to generate scenarios and automate the applications of the different heuristics to the scenarios:

```python
def get_scenarios(moves=3, n=200):
    scenarios = []
    for i in range(n):
        puzzle = eightpuzzle.createRandomEightPuzzle(moves)
        elements = []
        for row in puzzle.cells:
            for cell in row:
                elements.append(cell)
        scenarios.append(tuple(elements))
    return scenarios

def save_csv(scenarios, file_name):
    with open(file_name, 'w', newline='') as csvfile:
        writer = csv.writer(csvfile)
        for scenario in scenarios:
            writer.writerow(scenario)
    print(f"Scenarios saved to {file_name}.")

def load_scenarios(filename='scenarios2.csv'):
    scenarios = []
    with open(filename, 'r') as csvfile:
        reader = csv.reader(csvfile)
        for row in reader:
            scenario = list(map(int, row))
            scenarios.append(scenario)
    print(f"Loaded {len(scenarios)} scenarios from {filename}.")
    return scenarios

def run_search(problem, search_method, heuristic=None):
    try:
        heuristic_name = heuristic.__name__ if heuristic else 'None'
        print(f"Running search: {search_method.__name__}, heuristic: {heuristic_name}")

        if heuristic:
            result = search_method(problem, heuristic)
        else:
            result = search_method(problem)

        if len(result) == 4:  # Assuming result format is (path, expanded_nodes, fringe_size, depth)
            return result
        else:
            raise ValueError("Incorrect return format")

    except Exception as e:
        print(f"Error during search with {search method.  name  }: {e}")
```

```python
def compare_methods(scenarios, methods, output_file='results2.csv'):
    results = []
    print(f"Comparing search methods with {len(scenarios)} scenarios.")

    for index, scenario in enumerate(scenarios):
        state = eightpuzzle.EightPuzzleState(scenario)  # Adjust for your problem's state initialization
        problem = eightpuzzle.EightPuzzleSearchProblem(state)  # Adjust for your problem definition

        for method_name, method_details in methods.items():
            search_method = method_details['method']
            heuristic = method_details.get('heuristic')

            path, expanded_nodes, fringe_size, depth = run_search(problem, search_method, heuristic)

            # Convert the path to a string representation, if it's not already
            path_str = '->'.join(str(step) for step in path) if path else "No Path"

            results.append({
                'scenario': f"Scenario {index + 1}",
                'method': method_name,
                'expanded_nodes': expanded_nodes,
                'fringe_size': fringe_size,
                'depth': depth,
                'path': path_str  # Including path in the results
            })

    with open(output_file, 'w', newline='') as csvfile:
        fieldnames = ['scenario', 'method', 'expanded_nodes', 'fringe_size', 'depth', 'path']
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        for result in results:
            writer.writerow(result)

    print(f"Results written to {output_file}.")
```

After generating 100 different scenarios for the 8-puzzle and applying the different search methods to each one of these scenarios and recording the expanded nodes, fringe size, depth and path, we get the following results (please refer to the scenarios2.csv and results2.csv files in the folder to get the results of all 200 scenarios):



```
scenario,method,expanded_nodes,fringe_size,depth,path
Scenario 1,BFS,30,18,4,left->left->up->up
Scenario 1,DFS,23461,203,158,left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->right->right->up->left->left->down
Scenario 1,UCS,30,18,4,left->left->up->up
Scenario 1,A*,29,18,4,left->left->up->up
Scenario 1,h3,4,4,4,left->left->up->up
Scenario 2,BFS,6,8,2,up->left
Scenario 2,DFS,2639,203,160,right->down->left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->up->right->right->down
Scenario 2,UCS,6,8,2,up->left
Scenario 2,A*,5,8,2,up->left
Scenario 2,h3,2,5,2,up->left
Scenario 3,BFS,80,67,6,up->right->down->left->up->left
Scenario 3,DFS,106330,203,164,right->down->left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->up->right->right->do
Scenario 3,UCS,80,67,6,up->right->down->left->up->left
Scenario 3,A*,79,63,6,up->right->down->left->up->left
Scenario 3,h3,7,9,6,up->right->down->left->up->left
Scenario 4,BFS,6,8,2,up->left
Scenario 4,DFS,2639,203,160,right->down->left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->up->right->right->down
Scenario 4,UCS,6,8,2,up->left
Scenario 4,A*,5,8,2,up->left
Scenario 4,h3,2,5,2,up->left
Scenario 5,BFS,89,64,6,right->right->up->left->left->up
Scenario 5,DFS,147763,203,6,right->right->up->left->left->up
Scenario 5,UCS,89,64,6,right->right->up->left->left->up
Scenario 5,A*,88,63,6,right->right->up->left->left->up
Scenario 5,h3,6,7,6,right->right->up->left->left->up
Scenario 6,BFS,17,18,4,up->up->left->left
Scenario 6,DFS,32,43,32,left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->right->right->up->left->left
Scenario 6,UCS,17,18,4,up->up->left->left
Scenario 6,A*,16,18,4,up->up->left->left
Scenario 6,h3,4,4,4,up->up->left->left
Scenario 7,BFS,84,67,6,up->right->down->left->left->up
Scenario 7,DFS,70872,203,168,right->down->left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->up->right->right->dow
Scenario 7,UCS,84,67,6,up->right->down->left->left->up
Scenario 7,A*,83,63,6,up->right->down->left->left->up
Scenario 7,h3,8,10,6,up->right->down->left->left->up
Scenario 8,BFS,64,46,6,up->left->down->left->up->up
Scenario 8,DFS,85844,203,154,left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->right->right->down->left->left->up->right->right->up->left->left->down
Scenario 8,UCS,64,46,6,up->left->down->left->up->up
Scenario 8,A*,63,44,6,up->left->down->left->up->up
Scenario 8,h3,6,7,6,up->left->down->left->up->up
Scenario 9,BFS,1,2,0,No Path
Scenario 9,DFS,0,0,0,No Path
```

```
Results written to results2.csv.
        expanded_nodes  fringe_size   depth
method
A*               44.99        33.86    4.00
BFS              45.99        35.02    4.00
DFS           43562.93       152.76  111.76
UCS              45.99        34.96    4.00
h3                4.41         5.80    4.00
```

We notice that the uniformed searches have a bigger number of expanded nodes and fringe size. While A* offers a result close to the one found in BFS, we notice that DFS gives a large amount in all 3 of the categories. Overall, the Manhattan distance heuristic remains the best with results 10 times smaller than the other search methods.

V. Conclusion:

In summary, this research endeavor scrutinized four heuristic functions aimed at enhancing the efficiency of addressing the Eight Puzzle problem: Misplaced Tiles Count, Sum of Euclidean Distances, Sum of Manhattan Distances, and Number of Tiles Out of Row and Column. Each heuristic exhibits distinct strengths and limitations, necessitating additional research to discern the most efficacious approach. The findings of this study contribute valuable insights to the realm of artificial intelligence

and offer guidance for the development of informed search algorithms tailored for combinatorial search problems.