SCHOOL OF SCIENCE & ENGINEERING

SPRING 2024

CSC 5356 01 Data Engineering and Visualization

Project#1
Eight-Puzzle Heuristics Study

Realized by:

Noura Ogbi
Charles Weitzenberg
Chaimaa Aissi

Supervised by:

Prof. Tajjedine Rachidi

Table of Contents

I. Introduction ..... **Error! Bookmark not defined.**
1. Background and Objectives 3
II. Literature Review 3
1. 8-Puzzle Game Overview 3
2. Heuristic Algorithms 5
3. Efficient Search Techniques 6
III. Methodology 6
1. Heuristic Selection 7
a. h1: Number of Misplaced Tiles 7
b. h2: Sum of Euclidean Distances of the Tiles from Their Goal Positions 7
c. h3: Sum of Manhattan Distances of the Tiles from Their Goal Positions 7
d. h4: Number of Tiles Out of Row + Number of Tiles Out of Column 7
2. Implementation Details 7
a. Search Algorithms 7
IV. Tasks 11
1. Task 1: Implement 4 different heuristics h1, h2, h3, h4 for the 8puzzle 11
2. Task 2: Comparing heuristics 15
3. Task 3: Overall comparison between strategies 21
V. Conclusion 23
References 23

I. Introduction

1. Background and Objectives

In the field of problem solving the 8-puzzle has long been a captivating challenge, for both puzzle enthusiasts and computer scientists. In this project report we delve into the world of search techniques and heuristics focusing on how they're implemented and assessed within the context of the 8-puzzle game.

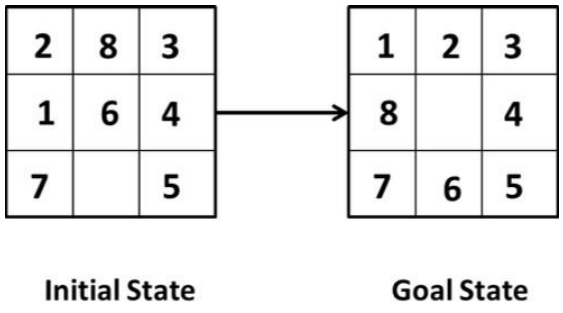
The 8 puzzle presents a problem, that consists of a 3x3 grid filled with eight tiles and one empty space requiring us to rearrange the tiles from a disordered state to a specified goal configuration. Our project revolves around algorithms that utilize estimates to guide our search for optimal solutions. These heuristics provide insights into determining the remaining cost needed to reach the goal state helping us prioritize promising paths over less fruitful ones.

Throughout our report we will showcase applications of heuristics and thoroughly evaluate their performance across different scenarios in the 8-puzzle. By doing our aim is to demonstrate how powerful heuristics are in solving problems while emphasizing their significance in artificial intelligence and algorithmic design. This project aims to uncover strategies for conquering the 8- puzzle shedding light on principles behind heuristic driven problem solving.

II. Literature Review

1. 8-Puzzle Game Overview

The 8 puzzle consists of eight numbered, movable tiles set in a 3x3 frame. One cell of the frame is always empty thus making it possible to move an adjacent numbered tile into the empty cell. Such a puzzle is illustrated in following diagram.[1] (dpthegrey, 2020)



[1] dpthegrey. (2020, June 30). Retrieved from Medium: https://medium.com/@dpthegrey/8-puzzle-problem- 2ec7d832b6db

The program aims to transform the initial setup into the target configuration. To find a resolution, we need to devise a coherent series of actions, such as "move tile 5 right," "shift tile 7 left," and "move tile 6 downward," among others.

The 8-puzzle problem is a mind-bending challenge that needs to arrange eight tiles in a 3X3 grid. The puzzle has three elements: the problem states, the moves can be made, and the goal. Each tile configuration represents a state, and the set of all possible configurations in the problem space is vast, with over 3.62 million different options. Describing the 8-puzzle with a straight forward description involves creating a 3X3 matrix, where each column represents a row, and each row represents a column. The first number in each cell of the matrix indicates the position of the first tile in the corresponding row or column, while the second number represents the second tile, and so on. The initial state of the 8-puzzle would be represented in the global database with this description. Instead of using a 3X3 matrix, any data type can be used to describe the initial problem state. Some common data structures include arrays, lists, and maps.

A move transforms one problem state into another state.

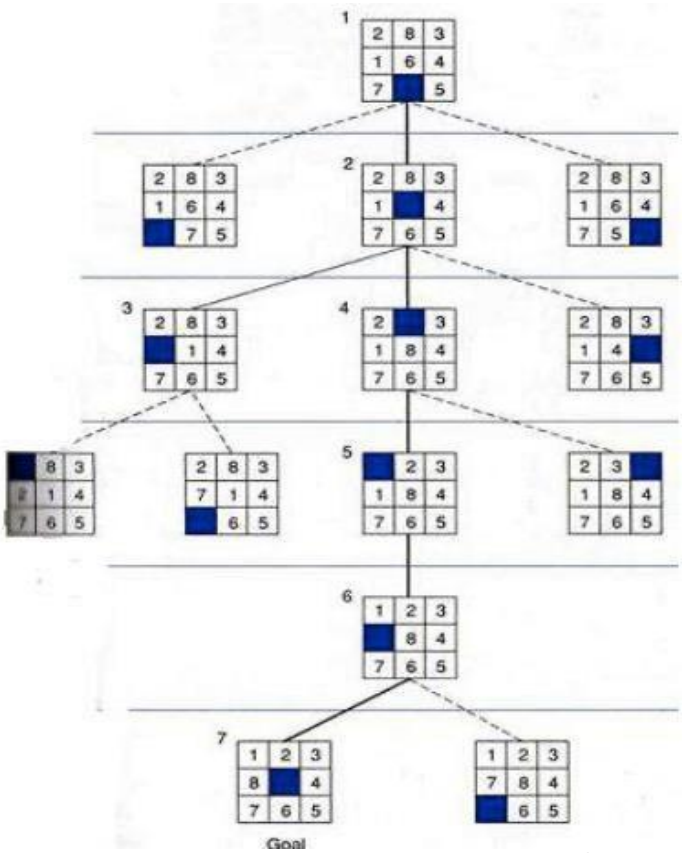
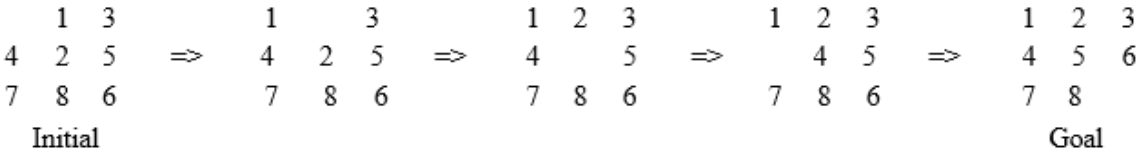


Figure 1: Solution of 8 Puzzle problem[2]

[2] dpthegrey. (2020, June 30). Retrieved from Medium: https://medium.com/@dpthegrey/8-puzzle-problem- 2ec7d832b6db

The 8-puzzle is a puzzle that can be solved by using four movements. These movements involve shifting a space also called a "blank," either horizontally diagonally to the left or to the right. The puzzle's behavior is determined by a set of rules that dictate how it changes when these moves are made. To win the game all spaces in the puzzle must be covered with tiles. Solving the puzzle involves employing a control strategy to identify moves and applying the rules to describe its state. By using this control strategy, we can find a solution by following a sequence of applied rules until we reach our goal. Essentially, the control strategy acts as a problem solver that explores move combinations until the desired outcome is achieved.



2. Heuristic Algorithms

Heuristic algorithms have a role in tackling problems, such as the 8-puzzle. They offer strategies that guide the search process effectively. These algorithms utilize functions to estimate the distance or cost between the state and the desired goal state.

The aim of heuristics is to strike a balance between efficiency and achieving a solution. In the realm of the 8 puzzle algorithms strive to gauge the proximity or distance between a puzzle arrangement and the desired end state. There are employed heuristics, such as the Manhattan distance misplaced tiles and pattern databases. Each heuristic provides a viewpoint on the problem impacting the efficiency and efficacy of search algorithms.

It is important to have an understanding of heuristic algorithms and their characteristics in order to choose the right techniques for solving the 8-puzzle. In this section of the literature review we will explore these methods, in detail examining their principles, benefits and constraints.

Principles:

- **Heuristic Functions:** Heuristic algorithms use functions that estimate the distance or cost, from the state to the goal state. These functions are designed to give an idea of how much effort is left to reach a solution. In the 8 puzzle functions include measuring the Manhattan distance and counting misplaced tiles.
- **Informed Search**: Heuristic algorithms guide the search process by evaluating each state's potential. Informed search techniques like A* combine estimates, with the cost of reaching a state from the state prioritizing states with lower estimated costs.
- **Admissibility and Consistency**; Heuristic functions can be admissible (never overestimating the cost) and consistent (satisfying the triangle inequality). Admissible heuristics guarantee that the search algorithm will find a solution while consistency can further enhance efficiency.

Benefits:

- **Efficiency**: Heuristics play a role in reducing the search space enabling efficient exploration of potential solutions. By focusing on states heuristic algorithms often discover solutions swiftly compared to uninformed search methods.
- **Optimality**: When an admissible heuristic is combined with an optimal search algorithm such as A* heuristics, ensure that the solution obtained is guaranteed to be optimal providing the path to reach the desired goal state.
- **Applicability**: One of the advantages of heuristics is their versatility; they can be customized to suit problem domains. For instance, in the case of the 8 puzzle different heuristics can capture aspects of the problem making them adaptable to scenarios.

Constraints:

- **Heuristic Accuracy**: The effectiveness of algorithms depends on how the heuristic function accurately guides the search. If the heuristic is either too optimistic or pessimistic it can lead to results resulting in solutions or increased computational expenses.
- **Computation and Memory**: Certain types of functions, those based on pattern databases or complex calculations can be computationally demanding and require a significant amount of memory. This can limit their practicality in environments, with resources.
- **Heuristic-Search Trade-off**: There's usually a trade-off between the quality of the heuristic and the computational effort needed. More accurate heuristics may require resources, which could potentially impact real time applications.
- **Problem-Specificity**: Although heuristics can be customized for problems they might not work effectively in different domains. Creating heuristics often requires an understanding of the specific problem at hand.

3. Efficient Search Techniques

Efficient search techniques are fundamental in solving the 8-puzzle optimally and in a time frame that relies heavily on key elements. The search algorithms were chosen to have an impact on the demands and overall effectiveness when tackling this puzzle.

There are used search algorithms for the 8-puzzle, such as depth first search (DFS), breadth first search (BFS), A* search and various forms of informed search. A* search is particularly noteworthy because of its capability to effectively blend heuristics with search algorithms.

III. Methodology

In this section we will explain the method we used for our study. Our focus was on using techniques outlined in the document we received to help us find the solutions for the 8-puzzle problem. These techniques played a role in our approach.

1. Heuristic Selection

a. h1: Number of Misplaced Tiles

In this approach we assess the quantity of tiles that are not in their positions, in the present state. This is considered an approach because it's clear that every tile that is not in its position will need to be moved at least once.

b. h2: Sum of Euclidean Distances of the Tiles from Their Goal Positions

This heuristic calculates the distance between each tile and its respective goal position using the method. It is considered valid because with each move a tile can only move one step closer to its goal and the Euclidean distance is always equal to or less than the number of steps needed to relocate a tile to its intended position.

c. h3: Sum of Manhattan Distances of the Tiles from Their Goal Positions

In this approach we calculate the Manhattan distances, between each tile and its intended position. This method is considered valid because with every move a tile can only move one step closer to its desired destination.

d. h4: Number of Tiles Out of Row + Number of Tiles Out of Column

h4 is an admissible heuristic since every tile that is out of column or out of row must be moved at least once and every tile that is both out of column and out of row must be moved at least twice.[3]

2. Implementation Details

In this section we will describe the details of how we implemented the search algorithms that're available in the search.py module.

a. Search Algorithms

Depth-First Search (DFS)

In this code snippet the depth FirstSearch function acts like an explorer. Just imagine giving it a puzzle. It becomes curious and eager to solve it! To accomplish this task, it utilizes a technique known as Depth First Search. You can think of it as navigating through a maze taking one step at a time and going deep, as possible before retracing its steps if it encounters an end.

[3] Dhesi, A. (n.d.). *Solving the 8-Puzzle using A* Heuristic Search.* Retrieved from Indian Institute of Technology Kanpur: https://cse.iitk.ac.in/users/cs365/2009/ppt/13jan_Aman.pdf

To keep track of the paths it has taken the explorer carries a stack. Similar to a backpack, known as the frontier. This stack holds the nodes that need to be explored. As the exploration progresses the explorer maintains a record of its visited locations in a list called explored nodes.

Ultimately, upon completing its mission the function provides a set of instructions.

```
def depthFirstSearch(problem):
    """Search the deepest nodes in the search tree first."""

    #states to be explored (LIFO). holds nodes in form (state, action)
    frontier = util.Stack()
    #previously explored states (for path checking), holds states
    exploredNodes = []
    #define start node
    startState = problem.getStartState()
    startNode = (startState, [])

    frontier.push(startNode)

    while not frontier.isEmpty():
        #begin exploring last (most-recently-pushed) node on frontier
        currentState, actions = frontier.pop()

        if currentState not in exploredNodes:
            #mark current node as explored
            exploredNodes.append(currentState)

            if problem.isGoalState(currentState):
                return actions
            else:
                #get list of possible successor nodes in
                #form (successor, action, stepCost)
                successors = problem.getSuccessors(currentState)

                #push each successor to frontier
                for succState, succAction, succCost in successors:
                    newAction = actions + [succAction]
                    newNode = (succState, newAction)
                    frontier.push(newNode)

    return actions
```

Breadth-First Search (BFS)

In this code the breadth First Search function behaves like an explorer armed with a map. Just picture it being given a puzzle to solve – it's determined to find the solution! To accomplish this, it employs a technique called Breadth First Search, which involves scanning an area row by row of diving deep right away.

The explorer maintains a queue, which's like a line of nodes waiting to be investigated to people waiting in line. This helps ensure that things are done in a manner examining one step at a time. To prevent getting trapped in loops it keeps track of where it has been by using a set called explored nodes like marking places visited on the map.

After completing its exploration, the function presents instructions, a sequence of actions comparable to footsteps, on a path that guides directly towards solving the puzzle and achieving the desired goal.

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""

    #to be explored (FIFO)
    frontier = util.Queue()

    #previously expanded states (for cycle checking), holds states
    exploredNodes = []

    startState = problem.getStartState()
    startNode = (startState, [], 0) #(state, action, cost)

    frontier.push(startNode)

    while not frontier.isEmpty():
        #begin exploring first (earliest-pushed) node on frontier
        currentState, actions, currentCost = frontier.pop()

        if currentState not in exploredNodes:
            #put popped node state into explored list
            exploredNodes.append(currentState)

            if problem.isGoalState(currentState):
                return actions
            else:
                #list of (successor, action, stepCost)
                successors = problem.getSuccessors(currentState)

                for succState, succAction, succCost in successors:
                    newAction = actions + [succAction]
                    newCost = currentCost + succCost
                    newNode = (succState, newAction, newCost)

                    frontier.push(newNode)

    return actions
```

A* Search

Think of the A* Search function, in this code as a pathfinder. It's like a traveler who loves solving puzzles. If there's a tool called a heuristic function it can use that to make its journey even more efficient.

The function follows a strategy called the A* search algorithm, which's similar to choosing paths by considering both the distance already traveled and an educated guess of how much's left. Imagine it as a hiker selecting routes based on both mileage and a rough map of the trail.

To maintain a record of its observations it employs a priority queue, akin to organizing tasks according to their significance. This approach assists in directing its attention towards the avenues initially much like delving into the most captivating enigmas within a narrative.

To avoid getting stuck in loops the system keeps track of its locations and the associated expenses similar to marking important milestones on a trip. This valuable information is stored in the explored nodes dictionary analogous to jotting down details in a travel journal.

Upon completion of its mission, the function provides a set of directions, a series of steps akin to a crafted map, that effectively and cleverly leads directly to the destination efficiently solving the puzzle.

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""

    #to be explored (FIFO): takes in item, cost+heuristic
    frontier = util.PriorityQueue()

    exploredNodes = [] #holds (state, cost)

    startState = problem.getStartState()
    startNode = (startState, [], 0) #(state, action, cost)

    frontier.push(startNode, 0)

    while not frontier.isEmpty():

        #begin exploring first (lowest-combined (cost+heuristic) ) node on frontier
        currentState, actions, currentCost = frontier.pop()

        #put popped node into explored list
        currentNode = (currentState, currentCost)
        exploredNodes.append((currentState, currentCost))

        if problem.isGoalState(currentState):
            return actions

        else:
            #list of (successor, action, stepCost)
            successors = problem.getSuccessors(currentState)

            #examine each successor
            for succState, succAction, succCost in successors:
                newAction = actions + [succAction]
                newCost = problem.getCostOfActions(newAction)
                newNode = (succState, newAction, newCost)

                #check if this successor has been explored
                already_explored = False
                for explored in exploredNodes:
                    #examine each explored node tuple
                    exploredState, exploredCost = explored

                    if (succState == exploredState) and (newCost >= exploredCost):
                        already_explored = True

                #if this successor not explored, put on frontier and explored list
                if not already_explored:
                    frontier.push(newNode, newCost + heuristic(succState, problem))
                    exploredNodes.append((succState, newCost))

    return actions
```

Uniform Cost Search (UCS)

In this code, the uniform Cost Search function takes a problem as input and explores the search space using the Uniform Cost Search algorithm. It utilizes a priority queue (frontier) to ensure nodes are expanded in ascending order of their cumulative costs. Explored states and their costs are stored in the explored nodes dictionary to ensure efficient cycle checking. The function returns a sequence of actions leading to the goal state.[4]

[4] ChatGPT

```python
def uniformCostSearch(problem):
    """Search the node of least total cost first."""

    #to be explored (FIFO): holds (item, cost)
    frontier = util.PriorityQueue()

    #previously expanded states (for cycle checking), holds state:cost
    exploredNodes = {}

    startState = problem.getStartState()
    startNode = (startState, [], 0) #(state, action, cost)

    frontier.push(startNode, 0)

    while not frontier.isEmpty():
        #begin exploring first (lowest-cost) node on frontier
        currentState, actions, currentCost = frontier.pop()

        if (currentState not in exploredNodes) or (currentCost < exploredNodes[currentState]):
            #put popped node's state into explored list
            exploredNodes[currentState] = currentCost

            if problem.isGoalState(currentState):
                return actions
            else:
                #list of (successor, action, stepCost)
                successors = problem.getSuccessors(currentState)

                for succState, succAction, succCost in successors:
                    newAction = actions + [succAction]
                    newCost = currentCost + succCost
                    newNode = (succState, newAction, newCost)

                    frontier.update(newNode, newCost)

    return actions
```

IV. Tasks

1. Task 1: Implement 4 different heuristics h1, h2, h3, h4 for the 8puzzle

```python
def h3test(current_state, glstate):
    total_distance = 0
    # iterates through each tile
    for row in range(3):
        for col in range(3):
            current_tile = current_state.cells[row][col]
            if current_tile != 0:
                # iterates through the goal state to check how far away the current tile is from its proper state
                for goal_row in range(3):
                    for goal_col in range(3):
                        # if the current tile belongs in the current goal state, the distance from the tile is set as
                        # abs(X(current)-X(goal)) + abs(Y(current)-Y(goal))
                        if glstate.cells[goal_row][goal_col] == current_tile:
                            distance = abs(row - goal_row) + abs(col - goal_col)
                            total_distance += distance
    return total_distance
```

The h1test function calculates the h1 heuristic, which determines how many cells in the state match the goal state. It goes through each cell of the puzzle, compares the values in the state with those in the goal state and increases the count for every matching cell. The final count indicates how many cells in the state are correctly positioned according to the goal state. Subtracting this count, from 8 gives us the number of tiles that're not in their positions.

```
def h2test(current_state, glstate):
    total_distance = 0
    # Iterates through each tile
    for row in range(3):
        for col in range(3):
            current_tile = current_state.cells[row][col]
            if current_tile != 0:  # Skip the blank (empty) tile
                # iterates through the goal state to check how far away the current tile is from its proper state
                for goal_row in range(3):
                    for goal_col in range(3):
                        # subs the x and y coordinate of the goal state from the current state and finds the length of
                        # the hypotonuse (sqrt(a^2+b^2) = c)
                        if glstate.cells[goal_row][goal_col] == current_tile:
                            distance = math.sqrt((row - goal_row) ** 2 + (col - goal_col) ** 2)
                            total_distance += distance
    return total_distance

```

The h2test function computes the value for the 8-puzzle problem. The h2 heuristic is determined by summing up the distances between each tile and its target position. In this function we iterate through each tile in the state calculate its distance from its intended position in the goal state and then add up these distances to obtain the overall h2 heuristic value. This specific heuristic measures how far each tile is from its desired position continuously, by utilizing the Euclidean distance formula.

The function calculates the h3 heuristic value, which is the sum of Manhattan distances of tiles from their goal positions. In this approach for every tile in the state (except for the tile denoted by 0) it computes the Manhattan distance. This distance is determined by adding up the differences between the row and column indices of each tiles position and its desired position. The total h3 heuristic value is the sum of these Manhattan distances for all tiles in the current state. This method assesses how many total moves (both horizontally and vertically) are needed to relocate each tile from its spot to its intended location.

```
def h4test(current_state, glstate):
    out_of_row_count = 0
    out_of_col_count = 0

    for row in range(3): # iterates through each tile
        for col in range(3):
            current_tile = current_state.cells[row][col] # sets the current tile
            if current_tile != 0: # Skip the blank (empty) tile
                inrow = 0
                # Iterates though the column again to check if the current tile is in it, if so, adds one to the count
                for c in range(3):
                    if current_tile == glstate.cells[row][c]:
                        inrow = 1
                if inrow == 0:
                    out_of_row_count +=1
                # Iterates though the column again to check if the current tile is in it, if so, adds one to the count
                incol = 0
                for r in range(3):
                    if current_tile == glstate.cells[r][col]:
                        incol = 1
                if incol == 0:
                    out_of_col_count += 1

    return out_of_row_count + out_of_col_count

#/*=====End Change Task 1 =====*/
```

The function calculates the h4 heuristic value by considering the number of tiles that're not in their correct row or column positions. For each tile, in the state (excluding the tile represented by 0) it checks if the tile is in the correct row and column according to the goal state. If a tile is not in the row, we increment a counter called "out_of_col_count". Similarly, if a tile is not, in the column we increment another counter called "out_of_col_count". The final h4 heuristic value is obtained by adding up both counters, which represents the number of tiles that're out of their intended positions in terms of rows and columns according to the goal state. This heuristic helps us understand how many tiles need to be moved vertically to reach their desired positions.

```
-------------
| 3 | 1 | 2 |
-------------
| 6 |   | 7 |
-------------
| 8 | 4 | 5 |
-------------
Press return for the next state...

After 4 moves: down
h1: 5
h2: 6.414213562373095
h3: 7
h4: 6

-------------
| 3 | 1 | 2 |
-------------
| 6 | 4 | 7 |
-------------
| 8 |   | 5 |
-------------
Press return for the next state...

After 5 moves: left
h1: 5
h2: 5.414213562373095
h3: 6
h4: 6

-------------
| 3 | 1 | 2 |
-------------
| 6 | 4 | 7 |
-------------
|   | 8 | 5 |
-------------
Press return for the next state...

After 6 moves: up
h1: 4
h2: 4.414213562373095
h3: 5
h4: 5

-------------
| 3 | 1 | 2 |
-------------
|   | 4 | 7 |
-------------
| 6 | 8 | 5 |
-------------
Press return for the next state...
```

```
After 7 moves: right
h1: 5
h2: 5.414213562373095
h3: 6
h4: 6

-------------
| 3 | 1 | 2 |
-------------
| 4 |   | 7 |
-------------
| 6 | 8 | 5 |
-------------
Press return for the next state...

After 8 moves: right
h1: 5
h2: 5.0
h3: 5
h4: 5

-------------
| 3 | 1 | 2 |
-------------
| 4 | 7 |   |
-------------
| 6 | 8 | 5 |
-------------
Press return for the next state...

After 9 moves: down
h1: 4
h2: 4.0
h3: 4
h4: 4

-------------
| 3 | 1 | 2 |
-------------
| 4 | 7 | 5 |
-------------
| 6 | 8 |   |
-------------
Press return for the next state...

After 10 moves: left
h1: 3
h2: 3.0
h3: 3
h4: 3
```

```
-------------
| 3 | 1 | 2 |
-------------
| 4 | 7 | 5 |
-------------
| 6 |   | 8 |
-------------
Press return for the next state...

After 11 moves: up
h1: 2
h2: 2.0
h3: 2
h4: 2

-------------
| 3 | 1 | 2 |
-------------
| 4 |   | 5 |
-------------
| 6 | 7 | 8 |
-------------
Press return for the next state...

After 12 moves: left
h1: 1
h2: 1.0
h3: 1
h4: 1

-------------
| 3 | 1 | 2 |
-------------
|   | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
Press return for the next state...

After 13 moves: up
h1: -1
h2: 0.0
h3: 0
h4: 0

-------------
|   | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
Press return for the next state...
```

This output shows the sequence of moves made by the A* search algorithm on the eight-puzzle problem along with the values of different heuristic functions at each step.

['left', 'down', 'down', 'right', 'right', 'up', 'up', 'left', 'down', 'left', 'up']: represents the sequence of moves made by the A* search algorithm to reach the goal state. Each move corresponds to a tile being moved in the puzzle.

h1: 7

h2: 8.414213562373096

h3: 9

h4: 8

At the initial state, the heuristic values for different heuristic functions are computed. These values represent the estimated cost from the current state to the goal state. Lower values indicate a closer proximity to the goal state according to the heuristic.

After 1 move: left After 2 moves: down After 3 moves: down

...

After 11 moves: up

For each step in the A* path, the puzzle's state is displayed after applying the corresponding move. The heuristic values for different heuristics are recalculated at each step, showing how the estimated cost changes as the puzzle progresses toward the goal state.

2. Task 2: Comparing heuristics

The main objective of this task is to assess and compare how different admissible heuristics (h1 to h4) perform when solving the 8-puzzle problem. Admissible heuristics are estimations of the cost required to reach the desired end state from a starting point. They play a role in directing search algorithms towards finding the efficient and optimal solutions.

```
2
3   import csv
4   from eightpuzzle import (EightPuzzleState)
5
    1 usage
6   def load_csv(filename):
7
8       puzzle_data = []
9       with open(filename, 'r') as file:
10          csv_reader = csv.reader(file)
11          for row in csv_reader:
12              if len(row) == 9:
13                  puzzle = [int(cell) for cell in row]
14                  puzzle_data.append(puzzle)
15      return puzzle_data
```

The load_csv function reads puzzle configurations from a CSV file and returns them as a list.

```
1 usage
def create_puzzles_from_csv(puzzle_data):

    puzzles = []
    for puzzle_config in puzzle_data:
        puzzle = EightPuzzleState(puzzle_config)
        puzzles.append(puzzle)
    return puzzles
```

The create_puzzles_from_csv function takes the puzzle configurations and creates a list of EightPuzzleState objects.

```
247     n = 1
248     maxFringe = 0
249     maxDepth = 0
        n -= 1

        if (len(actions) > maxDepth):
            maxDepth = len(actions)
```

```
if problem.isGoalState(currentState):
    return [actions, maxFringe, maxDepth, len(exploredNodes)]
```

In order to retrieve the maximum fringe size, a counter is added into each search algorithm that will increase by one when a node is added to the fringe and decrease by one when a node is popped. If at any point the value of this counter is larger than the value of maxFringe, then maxFringe is set to the value and the loop continues.

In order to retrieve the maximum depth, an if statement checks if the current value of maxDepth is less than the length of the actions leading to this node. If so, maxDepth is set to the length of the path.

In order to get the total number of explored nodes, I return the size of the exploredNodes array.

The screenshots of code above exist in all four of the search algorithm functions in search.py, and return the respective variables to the eightpuzzle.py for processing.

```
if __name__ == '__main__':

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

    h1_values_Astar = []
    h2_values_Astar = []
    h3_values_Astar = []
    h4_values_Astar = []
    h3_values = []

    path_lengths_Astar = []
    path_lengths_bfs = []
    path_lengths_dfs = []
    path_lengths_ucs = []

    AstarMaxFringes = []
    bfsMaxFringes = []
    dfsMaxFringes = []
    ucsMaxFringes = []

    AstarMaxDepths = []
    bfsMaxDepths = []
    dfsMaxDepths = []
    ucsMaxDepths = []

    AstarExploredNodes = []
    bfsExploredNodes = []
    dfsExploredNodes = []
    ucsExploredNodes = []

    n = 0
    num = 100
    num_moves = 10
    # creates n amount of puzzles and appends the h1 to 4 values to the lists
```

To begin the main statement, all the necessary variables and arrays are initialized. These include the heuristic value arrays, the search algorithm data arrays, and the variables used in the while loop.

```
while n < num:

    puzzle = createRandomEightPuzzle(num_moves)
    problem = EightPuzzleSearchProblem(puzzle)

    path_Astar = search.aStarSearch(problem)
    path_lengths_Astar.append(len(path_Astar[0]))
    AstarMaxFringes.append(path_Astar[1])
    AstarMaxDepths.append(path_Astar[2])
    AstarExploredNodes.append(path_Astar[3])

    h1_values_Astar.append(search.h1test(puzzle, goal_state) - len(path_Astar))
    h2_values_Astar.append(search.h2test(puzzle, goal_state) - len(path_Astar))
    h3_values_Astar.append(search.h3test(puzzle, goal_state) - len(path_Astar))
    h4_values_Astar.append(search.h4test(puzzle, goal_state) - len(path_Astar))

    h3_values.append(search.h3test(puzzle, goal_state))

    path_bfs = search.breadthFirstSearch(problem)
    path_lengths_bfs.append(len(path_bfs[0]))
    bfsMaxFringes.append(path_bfs[1])
    bfsMaxDepths.append(path_bfs[2])
    bfsExploredNodes.append(path_bfs[3])

    path_dfs = search.depthFirstSearch(problem)
    path_lengths_dfs.append(len(path_dfs[0]))
    dfsMaxFringes.append(path_dfs[1])
    dfsMaxDepths.append(path_dfs[2])
    dfsExploredNodes.append(path_dfs[3])

    path_ucs = search.uniformCostSearch(problem)
    path_lengths_ucs.append(len(path_ucs[0]))
    ucsMaxFringes.append(path_ucs[1])
    ucsMaxDepths.append(path_ucs[2])
    ucsExploredNodes.append(path_ucs[3])

    n += 1
```

This while loop creates n amount of puzzles and runs every heuristic and search algorithm on each one, recording the information into an array.

```
# Calculate the average absolute differences
avg_diff_h1_Astar = statistics.mean(h1_values_Astar)
avg_diff_h2_Astar = statistics.mean(h2_values_Astar)
avg_diff_h3_Astar = statistics.mean(h3_values_Astar)
avg_diff_h4_Astar = statistics.mean(h4_values_Astar)

avg_h3 = statistics.mean(h3_values)
avg_Astar = statistics.mean(path_lengths_Astar)
avg_bfs = statistics.mean(path_lengths_bfs)
avg_dfs = statistics.mean(path_lengths_dfs)
avg_ucs = statistics.mean(path_lengths_ucs)

astarAvgMaxFringe = statistics.mean(AstarMaxFringes)
bfsAvgMaxFringe = statistics.mean(bfsMaxFringes)
dfsAvgMaxFringe = statistics.mean(dfsMaxFringes)
ucsAvgMaxFringe = statistics.mean(ucsMaxFringes)

astarAvgMaxDepth = statistics.mean(AstarMaxDepths)
bfsAvgMaxDepth = statistics.mean(bfsMaxDepths)
dfsAvgMaxDepth = statistics.mean(dfsMaxDepths)
ucsAvgMaxDepth = statistics.mean(ucsMaxDepths)

astarAvgExploredNodes = statistics.mean(AstarExploredNodes)
bfsAvgExploredNodes= statistics.mean(bfsExploredNodes)
dfsAvgExploredNodes = statistics.mean(dfsExploredNodes)
ucsAvgExploredNodes = statistics.mean(ucsExploredNodes)
```

The code above takes the mean of the arrays that were recorded in the while loop. This includes the average difference of each heuristic from the average length of A star, as well as the average fringe size, maximum depth, and number of explored nodes for each search algorithm.

```
print("Average maximum size of the fringe for each search algorithm using " + str(num) + " puzzles with " + str(
    num_moves) + " random moves")
print("-------------------------------------------------------------")
print("A Star: " + str(astarAvgMaxFringe))
print("Breadth First Search: " + str(bfsAvgMaxFringe))
print("Depth First Search: " + str(dfsAvgMaxFringe))
print("Uniform Cost Search: " + str(ucsAvgMaxFringe))
print()

print("Average maximum depth for each search algorithm using " + str(num) + " puzzles with " + str(
    num_moves) + " random moves")
print("-------------------------------------------------------------")
print("A Star: " + str(astarAvgMaxDepth))
print("Breadth First Search: " + str(bfsAvgMaxDepth))
print("Depth First Search: " + str(dfsAvgMaxDepth))
print("Uniform Cost Search: " + str(ucsAvgMaxDepth))
print()

print("Average number of expanded nodes for each search algorithm using " + str(num) + " puzzles with " + str(
    num_moves) + " random moves")
print("-------------------------------------------------------------")
print("A Star: " + str(astarAvgExploredNodes))
print("Breadth First Search: " + str(bfsAvgExploredNodes))
print("Depth First Search: " + str(dfsAvgExploredNodes))
print("Uniform Cost Search: " + str(ucsAvgExploredNodes))
print()

print("Average path length for h3 and each algorithm using  " + str(num) + " puzzles with " + str(
    num_moves) + " random moves")
print("-------------------------------------------------------------")
print("h3: " + str(avg_h3))
print("A Star: " + str(avg_Astar))
print("Breadth First Search: " + str(avg_bfs))
print("Depth First Search: " + str(avg_dfs))
print("Uniform Cost Search: " + str(avg_ucs))
print()
```

The code above prints out the data found in a neat and tidy manner.

To recap, the main function operates by first initializing a large number of arrays. It then runs a while loop which generates a set number of puzzles, each with a set number of random moves made to them.

For each puzzle, the loop calculates the h1 (the number of misplaced tiles), h2 (sum of Euclidian distance of tiles from their goal positions), h3 (sum of Manhattan distance from their goal position), and h4 (the number of tiles out of the rows plus the number of tiles out of the columns). Then, each search algorithm (A star, breadth first search, depth first search, and uniform cost search) is run and the average fringe size, maximum depth, and number of explored nodes are added to their respective arrays.

After the loop is completed, the average of each array is taken. This should give us a good comparison for how each search function and heuristic performed. Finally, these values are sent to the print functions where they are outputted to the terminal.

```
average absolute difference of heuristics from A* path length using 100 puzzles with 10 random moves
h1: 0.58
h2: 0.26987806691180244
h3: 0.1
h4: 0.1

average h3 length: 3.6
average Astar length: 3.7
average bfs length: 3.7
average ucs length: 3.7

Would you like to run scenarios.csv? (yes/no): yes
-------------
| 3 | 1 | 2 |
-------------
| 6 |   | 5 |
-------------
| 7 | 4 | 8 |
-------------
A* found a path of 4 moves.
['down', 'left', 'up', 'up']
-------------
| 3 | 1 | 2 |
-------------
| 6 |   | 5 |
-------------
| 7 | 4 | 8 |
-------------
A* found a path of 4 moves.
['down', 'left', 'up', 'up']
-------------
| 1 | 4 | 2 |
-------------
| 3 |   | 5 |
-------------
| 6 | 7 | 8 |
-------------
A* found a path of 2 moves.
['up', 'left']
-------------
| 1 | 4 | 2 |
-------------
| 3 |   | 5 |
-------------
| 6 | 7 | 8 |
-------------
A* found a path of 2 moves.
['up', 'left']
-------------
| 1 | 4 | 2 |
-------------
| 3 | 7 | 5 |
-------------
| 6 | 8 |   |
-------------
A* found a path of 4 moves.
['left', 'up', 'up', 'left']
-------------
```

3. Task 3: Overall comparison between strategies

After gathering data and examining each search algorithm with extreme scrutiny, we have arrived at a conclusion. Out of all the possible search algorithms, there is one who reigns supreme. This algorithm is A-Star.

```
Average maximum size of the fringe for each search algorithm using 100 puzzles with 10 random moves
------------------------------------------------------------
A Star: 45.73
Breadth First Search: 78.64
Depth First Search: 13.44
Uniform Cost Search: 78.64

Average maximum depth for each search algorithm using 100 puzzles with 10 random moves
------------------------------------------------------------
A Star: 3.98
Breadth First Search: 3.98
Depth First Search: 8.56
Uniform Cost Search: 3.98

Average number of expanded nodes for each search algorithm using 100 puzzles with 10 random moves
------------------------------------------------------------
A Star: 183.82
Breadth First Search: 70.07
Depth First Search: 198.95
Uniform Cost Search: 70.07

Average path length for h3 and each algorithm using  100 puzzles with 10 random moves
------------------------------------------------------------
h3: 3.86
A Star: 3.98
Breadth First Search: 3.98
Depth First Search: 3.3
Uniform Cost Search: 3.98
```

Despite the above data showing that the depth first search algorithm is faster than A-Star, depth first is actually the worst performing program out of the 4. This is because many of the paths that depth first can follow to attempt to solve the search problem simply do not end. If left to its own devices, depth first search that was given would continue to fall down the rabbit hole of possible puzzle combinations until the end of time. To solve this issue, I added a limiter that will prevent the children of a node from being added to the fringe should the path be any longer than 10. Because of this, often times depth first will return a path that does not actually lead to the A* star is the most effective algorithm because it takes into consideration the probability that the path which it takes will lead to the goal. We can see this working in the statistics that were retrieved from the process running. As you can see from the average maximum size of the fringe in each algorithm, A Star runs with almost half of the amount of nodes in it's fringe compared to BFS and UCS . This is due to pruning. A Star is able to avoid paths which are

unlikely to lead to it's goal, while breadth first search simply tests the nodes that were added to the fringe first and uniform cost search simply tests the node with the lowest cost first.

V. Conclusion

In concluding our project, we have explored the captivating realm of the 8-puzzle game and its solutions guided by heuristics. By applying and evaluating heuristics (h1, h2, h3, h4) we have uncovered the effectiveness of these approaches in maneuvering puzzles. Through comparison and analysis, we identified the efficient heuristic highlighting its superiority over conventional search techniques such as BFS and DFS.

This journey has not just been about solving puzzles; it's been a discovery of intelligent search strategies. We've learned that heuristics are not just algorithms; they're guiding principles that illuminate the path toward optimal solutions. This project has equipped us with valuable insights, showcasing the power of smart algorithms in real-world problem-solving.

References

Dhesi, A. (n.d.). Solving the 8-Puzzle using A* Heuristic Search. Retrieved from Indian Institute of Technology Kanpur: https://cse.iitk.ac.in/users/cs365/2009/ppt/13jan_Aman.pdf

dpthegrey. (2020, June 30). Retrieved from Medium: https://medium.com/@dpthegrey/8-puzzle- problem-2ec7d832b6db