COMP 469 ARTIFICIAL INTELLIGENCE

Professor Taehyung (George) Wang.

The 8-puzzle problem

Team Members:

Pietro Cicciari,
Mario Choto,
Israel Sanchez,
Raj Kumar,
Armon Lee,
Venkat Sairam Ravala

Table of Contents:

Introduction

In this project, we focused on solving the 8-puzzle problem, a classic puzzle that involves arranging its pieces on a 3 * 3 board. The puzzle starts with 8 numbered tiles and one empty space. The goal is to rearrange the tiles so that they are in numerical order from 1 to 8, with the empty space in the bottom-right corner.

To solve this puzzle, we used different algorithms, each of them popularly known for its efficiency in terms of time and complexity. These algorithms help us find the steps needed to reach the goal state from any given starting configuration. By comparing these methods, we can determine which one works best for solving the 8-puzzle problem and provide a clear conclusion on how to approach this type of problem and even a problem with a larger amount of data.

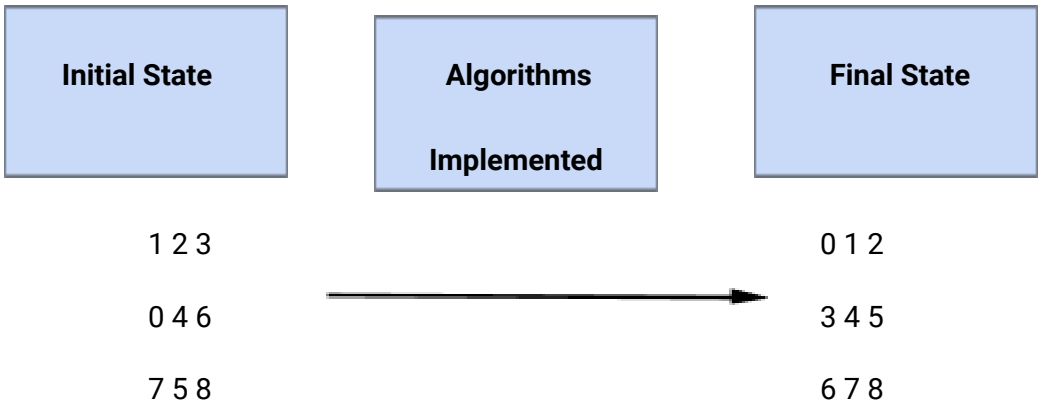
Formalizing Goal and Problem

The 8-puzzle problem involves a 3x3 grid with 8 numbered pieces and one empty space. The pieces are numbered from 1 to 8 or similar types of chronological topics, and the empty space allows the tiles to be moved around the grid.

Goal

The objective of the 8-puzzle problem is to rearrange the tiles to achieve a specific target configuration known as the "goal state."

Here's what the game looks like in a nutshell:



Agent Type and Justification

We chose the agent best fits for the task, **Goal-Based Agent**. This type of agent perfectly fits in because the main idea is to achieve a specific target configuration (goal state) by rearranging the pieces. Goal-based agents can carry out and execute actions to reach a specific goal, which aligns perfectly with the requirements of the 8-puzzle problem we are working on.

PEAS Description Of The Agent

To design a rational agent, we used the PEAS (Performance measure, Environment, Actuators, and Sensors) framework to specify the agent's functioning and measure its capacity:

1. **Performance Measure**:
    - Number of moves taken to reach the goal state.
    - Time taken to reach the goal state.
    - Memory usage during the process.
    - Optimality of the solution (finding the least-cost solution).
2. **Environment**:
    - A 3 * 3 board (grid in this case) representing the 8-puzzle board.
    - The environment is fully observable, deterministic, sequential, static, discrete, and known.
3. **Actuators**:
    - The ability to move tiles (up, down, left, right).
4. **Sensors**:
    - The current configuration of the puzzle, for instance, the positions of the tiles and the empty space.

Properties:

- **Fully Observable vs. Partially Observable**: Fully observable as the entire state of the grid is known at all times.
- **Deterministic vs. Stochastic**: Deterministic as each action has a predictable outcome.
- **Episodic vs. Sequential**: Sequential as each action depends on the previous sequence of actions.
- **Static vs. Dynamic**: Static as the puzzle does not change unless the agent makes a move.
- **Discrete vs. Continuous**: Discrete as there are a finite number of states and actions.
- **Known vs. Unknown**: Known as the rules and goal state are defined.

Formal Specifications of the Problem

- **Set of Possible States**:
  - All possible configurations of the 3 * 3 grid where each configuration is a unique way of arranging the pieces.
- **Initial State**:
  - A starting configuration of the puzzle provided to the agent to start off.
- **Goal State**:

  The desired target configuration after the agent's performance:
  0 1 2

  3 4 5

  6 7 8

- **Actions Available to the Agent**:
  - Move the empty space (represented always by 0) up, down, left, or right.
- **Transition Model**:
  - Defines the result of moving the empty space in a given direction:
    - If the empty space moves up, the tile above it moves down.
    - If the empty space moves down, the tile below it moves up.
    - If the empty space moves left, the tile to the left moves right.
    - If the empty space moves right, the tile to the right moves left.
- **Action Cost Function**:
  - Typically, each move has a uniform cost of 1. This simplifies the calculation and comparison of different paths to the goal state.
  -

2. Explore Search Solutions and Apply Search Algorithms

Exploring Solutions:

We explored each of the several algorithms that could carry out the task of the 8 Puzzle problem and came up with the following descriptions and a chart that allows us to visually see and even compare the functionality of each algorithm during the performance of the same task.

We used five main algorithms to solve the 8-puzzle problem:

1. **Breadth-First Search (BFS)**: This algorithm explores all possible moves level by level. It checks all positions that can be reached in one move before moving on to positions that can be reached in two moves, and so on.
2. **Depth-First Search (DFS)**: This algorithm explores as far as possible along one branch before backtracking. It goes deep into one possible move sequence until it can't go any further, then backtracks to try a different path.
3. **A* Search Algorithm**: This algorithm uses a heuristic to guide the search, which means it estimates how close each move is to the goal and prioritizes the closest ones.
4. **Greedy Best-First Search**: This algorithm always chooses the move that appears to be closest to the goal based on a heuristic.
5. **Iterative Deepening Search**: This algorithm combines the depth-first and breadth-first approaches, gradually increasing the depth limit until it finds a solution.

Algorithms Process For the 8 Puzzle Problem

Definitions Needed to Understand the Following Chart:

- **Completeness**: Indicates whether the algorithm is guaranteed to find a solution if there is one.
- **Cost Optimality**: Reflects whether the algorithm finds the least-cost path to the solution.
- **Time Complexity**: Measures the time taken by the algorithm to find the solution, typically in terms of the branching factor (b) and depth (d).
- **Space Complexity**: Measures the memory usage of the algorithm during its execution.

Algorithm	Completeness	Cost Optimality	Time Complexity	Space Complexity
BFS (Breadth-First Search)	Complete (will find a solution if it exists)	Finds shortest path if one exists	Exponential: O(b^d)	High: O(b^d)
DFS (Depth-First Search)	Not complete (may get stuck in loops)	May not find shortest path	Exponential: O(b^m)	Lower than BFS: O(bm)
A Search Algorithm*	Complete (will find a solution if it exists)	Often finds shortest path quickly	Exponential: O(b^d) with heuristic improvements	Depends on heuristic and state space
Greedy Best-First Search	Not complete (may miss the optimal path)	Not guaranteed to find shortest path	Exponential: O(b^m)	Depends on heuristic and state space
Iterative Deepening Search	Complete (will find a solution if it exists)	Finds shortest path if one exists	Exponential: O(b^d)	Moderate: O(bd)

4. Implement and Execute Code

BFS Algorithm Implementation:

The BFS algorithm implemented in Python operates by exploring all potential configurations of the puzzle from the initial state, expanding nodes level by level until the solution is found. Our Algorithm begins by prompting the user to provide an input of a series of 8 numbers that will constitute the puzzle. The puzzle is then evaluated by the program and all possible moves are generated from the initial state provided from the user. The algorithm then treats the generated puzzles as the new parent nodes and generates all possible moves off of those puzzles. This means that in a short amount of time thousands of nodes are generated each step, leading to the algorithm requiring large amounts of memory. During the node generation process the parent puzzle is checked to see if it is the goal state or if the puzzle has generated children before. Since BFS is a complete algorithm the goal state will eventually be found but massive amounts of memory may be required to find it.

The following pseudo code for the BFS Algorithm.

```
Pseudo Code BFS 8 Puzzle Problem

# Locate start tile function
def TileLoc()
        Loc = 0
        For i in puzzle
                If puzzle[i] = 0
                        Loc = i
return loc

#function to swap 0 tile with a tile above and return new puzzle
Def move up (puzzle, loc)
        If loc > 3
                Puzzle_Temp = Puzzle
                Temp_val = Puzzle_Temp[loc +3]
                Puzzle_Temp[loc] =  Temp_val
                Puzzle_Temp[loc +3] = 0
                Return puzzle_temp

# function to swap 0 tile with a tile below and return new puzzle
Def move down(puzzle, loc)
        If loc > 2
                Puzzle_Temp = Puzzle
                Temp = Puzzle_Temp[loc -3]
                Puzzle_Temp[loc] =  Temp
                Puzzle_Temp[loc - 3] = 0
                Return puzzle_temp


#function to swap 0 tile with a tile to the left and return new puzzle
Def move left(puzzle, loc)
        If loc != 0, 3, 6
                Puzzle_Temp = Puzzle
                Temp = Puzzle_Temp[loc - 1]
                Puzzle_Temp[loc] =  Temp
                Puzzle_Temp[loc  -  1] = 0
                Return puzzle_temp

# function to swap 0 tile with a tile to the right and return new puzzle
Def move right(puzzle, loc)
        If loc != 2, 5, 8
                Puzzle_Temp = Puzzle
                Temp = Puzzle_Temp[loc + 1]
                Puzzle_Temp[loc] =  Temp
                Puzzle_Temp[loc +1] = 0
                Return Puzzle_temp
```

BFS Pseudo Code (Cont.)

```
        if parent != up
                child.append(up)
                c=c+1
        if parent != down
                child.append(down)
                c=c+1
        if parent != left
                child.append(left)
                c=c+1
        if parent != right
                child.append(right)
                c=c+1
return c + prev_num_child, child, c

#loop to iterate through all possible moves searching for the correct solution
def BFS(puzzle)
flag = 0 #flag to indicate if solution was found
explored = []  # Intialize variables
Parent = puzzle  #Current Parent puzzle node
child = [] #child nodes
num_parents = 1
num_child = 0
node_per_p = 0
iteration = 0
while flag = 0
        for c in range(num_parents):
                puzzle_list = Parent
                if puzzle_list = [0, 1, 2, 3, 4, 5, 6, 7, 8]
                        Flag = 1
                        explored.append(puzzle_list)
                        break
                if Puzzle_list not in explored
                        num_child, child, node_per_p = search_node(puzzle_list, child,
num_child)
                        explore.append(puzzle_list)
                iteration+=1
                num_parents = len(child) / (9 + iteration)
                Parent = child
Print("Solution Found.")


Puzzle = [1, 2, 3, 0, 5, 8, 6, 7]
BFS(Puzzle)
```

BFS Code

```
import copy


# Create 8 number puzzle from input
def Gen_Puz():
    #Variables to handle input checking
    contentCheck = 0
    A = input()
    A_copy = copy.deepcopy(A)
    A_list = A_copy.split(" ")

    print("Enter the puzzle as a row")
    print("e.g 1 2 3 0 4 5 8 6 7 results in:")
    print("1 2 3")
    print("0 4 5")
    print("8 6 7")

    #loop to itterate through the list and ensure that it is all numbers from 0-8
    for i in range(0, len(A_list)):
        if A_list[i] == 0 or 1 or 2 or 3 or 4 or 5 or 6 or 7 or 8:
            contentCheck = contentCheck + 1

    #Check that the approptiate number of entries were provided
    if contentCheck == 9:
        B = [int(i) for i in A.split()]
        return B
    else:
        print("Invalid puzzle please restart and try again")



# Find the movable tile
def StartLocation(N):
    for i in range(9):
        if N[i] == 0:
            return i, # Return Starting position in list


# Move up function that creates a new node
def Up(b, N):
    U = copy.deepcopy(N)
    if b > 2:
        U[b] = U[b - 3]  # Swap Tiles
        U[b - 3] = 0
    return U  # Return New Node
```

BFS Code (Cont.)

```
# Move left function that creates a new node
def Left(b, N):
    L = copy.deepcopy(N)
    if b != 0 and b != 3 and b != 6:
        L[b] = L[b - 1]  # Swap Tiles
        L[b - 1] = 0
    return L  # Return New Node


# Move right function that creates a new node
def Right(b, N):
    R = copy.deepcopy(N)
    if b != 2 and b != 5 and b != 8:
        R[b] = R[b + 1]  # Swap Tiles
        R[b + 1] = 0
    return R  # Return New Node

# Add Node to Child list and find number of childs per parent
def AddNode(Parent, Pf, iteration, child, j):
    Pdir = Pf[9:len(Pf)]
    c = 0
    b = StartLocation(Parent)
    U = Up(b, Parent)
    D = Down(b, Parent)
    L = Left(b, Parent)
    R = Right(b, Parent)
    if Parent != U:  # Check if node is new or same
        child = child + U + Pdir  # Append new node and its action set
        child.append("U")
        c = c + 1
    if Parent != D:
        child = child + D + Pdir
        child.append("D")
        c = c + 1
    if Parent != L:
        child = child + L + Pdir
        child.append("L")
        c = c + 1
    if Parent != R:
        child = child + R + Pdir
        child.append("R")
        c = c + 1
    return c + j, child, c  # Return total child nodes, nodes, nodes per parent
```

BFS Code (Cont.)

```
# BFS algorithm to reach goal node
def BFS(N):
    inv = 0

    explored = []  # Intialize variables
    Parent = N
    child = []
    num_parents = 1
    num_child = 0
    node_per_p = 0
    iteration = 0
    f = 0
    while f == 0:  # Loop to evaluate tree levels
        child.clear()
        for c in range(num_parents):  # Loop to access different node in a particular level
            p_list = Parent[(0 + c * (9 + iteration)):(iteration + 9 + c * (9 + iteration))]
            p_slice = p_list[0:9]
            if p_slice == [0, 1, 2, 3, 4, 5, 6, 7, 8]:
                f = 1
                explored.append(p_slice)
                break
            if not p_slice in explored:  # Check for child node if parent not in explored
                num_child, child, node_per_p = AddNode(p_slice, p_list, iteration, child, num_child)
                explored.append(p_slice)
        iteration += 1
        num_parents = int(len(child) / (9 + iteration))
        Parent = copy.deepcopy(child)
        print(" Explored Nodes", len(explored))


    print(" Solution Found.")


A = Gen_Puz()  # Prompting user to provide puzzle input
BFS(A)  # Passing the puzzle to BFS Algorithm
```

Iterative Deepening Search (IDS).

This algorithm combines the completeness of the BFS algorithm with the memory efficiency of the DFS method. BFS and DFS are critical techniques for many applications in computer science related to searching, pathfinding, and network analysis. A choice is made based on the kind of application being used at hand; for instance, shortest paths could be sought using BFS and complete traversing through a graph using DFS. These algorithms provide an important foundation in understanding graph theory and the ability to solve problems built atop these concepts. First, IDS starts at a zero depth and increases the depth limit iteratively. At each level, it calls the function DLS, performing a Depth-Limited Search. DLS explores nodes up to some given depth, checking whether a node is the goal. Provided this, in case a goal node has been found, then the algorithm stops and returns the node; otherwise, it increments the depth and repeats the process until it reaches some goal.

IDS Algorithm:The method of the IDS keeps on increasing the depth limit and performs DLS until it finds a solution or runs through all possible depths. Thus, with this algorithm, IDS acquires the memory efficiency of Depth searching-First Search, together with the completeness of Breadth-First Search.

DLS and Recursive DLS: These are methods that look through the state space of the puzzle to a given depth limit by recursively creating and examining all of the possible moves from the current state. If it finds the goal state within the current depth, then it returns the solution path.

Execution Flow: The main() function initializes the puzzle, shuffles it to create a starting configuration, and then uses IDS to find a solution. The path to the solution and the number of states explored is then printed out.

This is a fine trade-off between completeness and memory usage, which makes this implementation appropriate for solving puzzles like the 8-puzzle.

IDS Pseudo Code:

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

Class EightPuzzle:
    // Methods like __init__, __eq__, __str__, _clone, _get_legal_moves,
    // _generate_moves, _generate_solution_path, shuffle, find, peek,
    // poke, swap are assumed to be implemented already.

    Method ids_solve():
        depth = 0
        total_move_count = 0

        Loop Forever:
            result, move_count = _dls(depth)
            total_move_count += move_count

            If result is not None:
                Return result, total_move_count  // Solution found

            Increment depth by 1

    Method _dls(limit):
        Return _recursive_dls(limit, 0, empty_path)

    Method _recursive_dls(limit, current_depth, path):
        If current state (self.adj_matrix) equals _goal_state:
            Return path, 1  // Solution found, return path and count 1 move

        If current_depth equals limit:
            Return None, 1  // Reached depth limit, return no solution and c

        move_count = 0

        For each move in _generate_moves():
            new_puzzle = Apply move to current state to generate new puzzle
            new_path = Append new_puzzle to path

            result, moves = new_puzzle._recursive_dls(limit, current_depth +

            move_count += moves

            If result is not None:
                Return result, move_count  // Solution found

        Return None, move_count  // No solution found within this depth

Method main():
    p = Instantiate EightPuzzle
    p.shuffle(20)  // Shuffle the puzzle to create a random state
    Print p  // Print the initial puzzle state

    path, count = p.ids_solve()  // Attempt to solve the puzzle with IDS

    If path is not None:
        For each state in path:
            Print state  // Print each step in the solution path

        Print "Solved with IDS exploring", count, "states"
    Else:
        Print "No solution found within the given depth limit"

// Execute the main function
Call main()
```

IDS Code:

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

class EightPuzzle:
    # (Keep the existing implementation of the EightPuzzle class here)
    # Methods: __init__, __eq__, __str__, _clone, _get_legal_moves,
    # _generate_moves, _generate_solution_path, shuffle, find, peek,
    # poke, swap should remain the same.

    def ids_solve(self):
        """
        Performs Iterative Deepening Search (IDS) for the goal state.
        Returns the solution path and the number of moves explored.
        """
        depth = 0
        move_count = 0

        while True:
            try:
                result, moves = self._dls(depth)
                move_count += moves
                if result:
                    return result, move_count
                depth += 1
            except RecursionError as e:
                print(f"Recursion depth exceeded at depth {depth}: {e}")
                break
            except Exception as e:
                print(f"An error occurred during IDS: {e}")
                break

    def _dls(self, limit):
        """
        Depth-Limited Search (DLS)
        limit - depth limit
        Returns the solution path if found within the limit, else None.
        """
        try:
            return self._recursive_dls(limit, 0, [])
        except Exception as e:
            print(f"An error occurred during DLS with limit {limit}: {e}")
            return None, 0
```

IDS Code (Cont.):

```
    def _recursive_dls(self, limit, depth, path):
        """
        Recursively performs depth-limited search.
        """
        try:
            if self.adj_matrix == _goal_state:
                return [path], 1

            if depth == limit:
                return None, 1

            move_count = 0
            for move in self._generate_moves():
                new_path = path + [move]
                result, moves = move._recursive_dls(limit, depth + 1, new_path)
                move_count += moves
                if result:
                    return result, move_count

            return None, move_count
        except Exception as e:
            print(f"An error occurred during recursive DLS at depth {depth}: {e}")
            return None, 0

def main():
    try:
        p = EightPuzzle()
        p.shuffle(20)
        print(p)

        path, count = p.ids_solve()
        if path:
            for state in path[0]:
                print(state)
            print(f"Solved with IDS exploring {count} states")
        else:
            print("No solution found within the given depth limit")
    except Exception as e:
        print(f"An error occurred in the main execution: {e}")

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

An Overview at the (DFS) Deep First Search Algorithm.

In this code the algorithm begins by allowing the user to input an initial puzzle configuration. It then goes and uses various functions to move the empty space around the grid by swapping it with adjacent tiles, exploring all possibilities and moves (up, down, left, right). The main part of the code, the DFS function, systematically explores these possible configurations by checking each one to see if it matches the initial goal. If the goal was found, then the algorithm stops; if not, it continues looking around until all possibilities are exhausted.

Benefits of Using This Code

The main benefit of using this code is its ability to systematically explore all potential solutions to the 8-puzzle problem. By using the DFS algorithm, the code can dive deep into potential solutions, which might quickly lead to the goal configuration. It's important to point out, that while DFS can be effective, it's not always the best method, as it can get stuck in deep but unproductive paths but despite this, the code can provide a solid foundation for understanding basic search algorithms and their application to problem-solving in computational tasks.

DFS Pseudo Code:

```
# Prompting user to provide puzzle input
# Create 8 number puzzle from input

# DFS Algorithm
   # Initialize stack that holds all nodes
   # Initialize explored array the keep track of explored nodes
   # Flag for goal node

   # Loop to evaluate nodes if stack is empty and Flag is not 1
      # Pop the last node from the stack → current
      # If current is goal node
         # flag is 1
         # break
      # If child node if parent not in explored
         # Find location of 0
         # Add new node based on all possible moves
         # Append valid moves to the stack
         # Append node in explored

   # If f is 1
      # print solution
   #else
      #print "no solution"
```

DFS Code

```
import copy

# Create 8 number puzzle from input
def Gen_Puz():
  checkA = []
  while checkA != [0, 1, 2, 3, 4, 5, 6, 7, 8]:
    print("Enter the puzzle as a row")
    print("e.g 1 2 3 0 4 5 8 6 7 results in:")
    print("1 2 3")
    print("0 4 5")
    print("8 6 7")
    try:
      A = [int(i) for i in input().split()]
      checkA = sorted(A)
      if checkA != [0, 1, 2, 3, 4, 5, 6, 7, 8]:
        print("Invalid input")
    except:
      print("Invalid input")

  return A

# Find the movable tile
def StartLocation(N):
  for i in range(9):
    if N[i] == 0:
      return i  # Return Starting position in list

# Move up function that creates a new node
def Up(b, N):
  U = copy.deepcopy(N)
  if b > 2:
    U[b] = U[b - 3]  # Swap Tiles
    U[b - 3] = 0
  return U  # Return New Node

# Move down function that creates a new node
def Down(b, N):
  D = copy.deepcopy(N)
  if b < 6:
    D[b] = D[b + 3]  # Swap Tiles
    D[b + 3] = 0
  return D  # Return New Node

# Move left function that creates a new node
def Left(b, N):
  L = copy.deepcopy(N)
  if b != 0 and b != 3 and b != 6:
    L[b] = L[b - 1]  # Swap Tiles
    L[b - 1] = 0
  return L  # Return New Node

# Move right function that creates a new node
def Right(b, N):
  R = copy.deepcopy(N)
  if b != 2 and b != 5 and b != 8:
    R[b] = R[b + 1]  # Swap Tiles
    R[b + 1] = 0
  return R  # Return New Node
```

DFS Code (Cont.)

```
# Add Node to Child list and find number of childs per parent
def AddNode(Parent, stack):
  b = StartLocation(Parent)
  U = Up(b, Parent) #Check possible moves
  D = Down(b, Parent)
  L = Left(b, Parent)
  R = Right(b, Parent)

  if Parent != U:  # Check if node is new or same
    stack.append(U) # Push into stack
  if Parent != D:
    stack.append(D)
  if Parent != L:
    stack.append(L)
  if Parent != R:
    stack.append(R)
  return stack  # Return the stack with new nodes

# DFS algorithm to reach goal node
def DFS(N):
  stack = [N] # Initialize stack
  explored = []  # Initialize explored
  f = 0 # Flag for goal node
  while stack and f == 0:  # Loop to evaluate nodes
    currentNode = stack.pop()  # Pop the last node from the stack
    if currentNode == [0, 1, 2, 3, 4, 5, 6, 7, 8]: #Check goal node
      f = 1
      explored.append(currentNode)
      break
    if not currentNode in explored:  # Check for child node if parent not in explored
      stack = AddNode(currentNode, stack) #Add new node
      explored.append(currentNode)
    print("Explored Nodes:", len(explored))

  if f == 1:
    print("Solution Found.")
  else:
    print("No Solution.")

A = Gen_Puz()  # Prompting user to provide puzzle input
DFS(A)  # Passing the puzzle to DFS Algorithm
```

The Greedy Best-First Search:

This algorithm first starts by making a random 8-puzzle that can actually be solved. To do that, it shuffles the tiles and checks if the puzzle is solvable by counting the pair of tiles that are out of order, also known as inversions. If the number of inversions is even, then the puzzle can be solved. If it's odd, it can't be solved. So, the program keeps shuffling until it finds a solvable puzzle.

Once there is a solvable puzzle, the Greedy Best-First Search algorithm starts. This algorithm's goal is to solve the puzzle by always choosing the move that appears to be the most promising. It determines the best move by calculating the "Manhattan distance" for each possible option. The Manhattan distance measures how far each tile is from its goal position, summing up all the distances for the tiles on the board.

The algorithm looks at all the possible moves from the current board configuration and picks the one with the lowest Manhattan distance, which means it's the closest to the goal. It doesn't consider the path cost; it only focuses on the move that seems the most promising at the moment. To prevent revisiting the same states, it keeps track of all the moves it has already tried.

This process continues, reaching its goal with each move, until the puzzle is either solved or there are no other moves available. In the 8-Puzzle case, since it is always solvable, there is no concern of the solution not being found.

When the algorithm finishes, it displays the number of sequences made and the amount of time it took to find the solution. This algorithm aims to get the solution as quickly as possible by choosing the move that appears most optimal in that instant.

Here is the pseudocode for the Greedy Best First Search:

```
Import heapq for priority queue operations
Import random for generating random numbers
Import time for measuring the time taken for the search

Function __init__(tiles, empty_position, steps=0, previous=None):
    set self.tiles to the current board configuration
    set self.empty_position to the position of the empty tile
    set self.steps to the number of moves made
    set self.previous to the previous PuzzleState
    calculate the hueristic value (Manhattan distance) using self.calculate_manhattan_distance()

Function calculate_manhattan_distance():
     compute the Manhattan distance heuristic
        initialize distance to 0
        for each row index from 0 to 2:
            for each column index from 0 to 2:
                get the value at the current position (row, column)
                if the value is not 0:
                    compute the goal row and column for this value
                    calculate the absolute difference between current and goal position
                    add these differences to distance
        return the total distance

Function find_possible_moves():
        generate all valid moves from the current state
        initialize an empty list called moves
        get the current position of the empty tile (x, y)
        define possible directions for moving the empty tile: Up, Down, Left, Right
        for each direction (dx, dy):
            calculate the new position (new_x, new_y) based on direction
            check if the new position is within board boundaries
            if valid:
                create a new board configuration by copying current tiles
                swap the empty tile with the tile at the new position
                create a new PuzzleState with updated configuration, empty position, and incremented step count
                add this new PuzzleState to the moveslist
        return the list of possible new PuzzleStates

Function __lt__(other):
        define comparison based on heuristic value for priority queue
        return true if this state's heuristic is less than the other's


Function is_goal_state():
        check if the current state matches the goal configuration
        return true if self.tiles equals the goal state [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

Function print_board():
        print the current board configuration
        for each row in self.tiles:
            convert each tile to string, replacing 0 with space for readability
            print the row
        print a blank line for separation

Function count_inversions(tiles):
    count the number of inversions in the board configuration
    flatten the 2D list of tiles into a 1D list, excluding the empty tile (0)
    initialize inversions count to 0
```

Greedy Best First Search pseudocode continued:

```
        for each pair of tiles (i, j) where i < j:
            if tile[i] > tile[j]:
                increment inversions count
        return the total number of inversions

    Function check_solvable(tiles):
        determine if the puzzle configuration is solvable
        calculate the number of inversions
        return true if the number of inversions is even (puzzle is solvable), otherwise return false

    Function create_random_puzzle():
        generate a random solvable puzzle configuration
        initialize a list called numbers with values from 0 to 8
        shuffle the numbers to create a random configuration
        convert the shuffled list into a 3x3 board (2D list)
        while the board is not solvable:
            shuffle numbers again and recreate the board
        find the position of the empty tile (0) in the board
        return the board and the position of the empty tile

    Function solve_puzzle(start_board):
        solve the puzzle using greedy best-first search algorithm
        find the position of the empty tile in the start_board
        create an initial PuzzleState with the start_board and empty tile position
        initialize a priority queue
        add the initial state to the priority queue
        initialize a set to keep track of visited states
        add the tuple representation of start_board to the visited set
        initialize move_counter to track the number of moves
        initialize nodes_explored to count the number of nodes processed
        initialize max_queue_size to track the maximum size of the priority queue
        record the start time for performance measurement
        initialize an empty list called solution_path for reconstructing the solution

        while the priority queue is not empty:
            update max_queue_size with the current size of the priority queue
            remove the state with the lowest heuristic value from the queue
            increment nodes_explored
            if the current state is the goal state:
                reconstruct the path from the goal state to the start
                reverse the path to get it from start to goal
                break from the loop
            for each move generated from the current state:
                convert the new state to a tuple for hashable representation
                if this new state has not been visited:
                    add it to the visited set
                    add it to the priority queue for further exploration

        record the end time for performance measurement

        if a solution path is found:
            print the solution path step by step
            for each state in solution_path:
                print the move number and the board configuration
                increment move_counter
            return a dictionary with solution details:
                -   steps to solve
                -   nodes explored
                -   maximum queue size
                -   time taken

        return none if no solution is found

    Main execution
    Generate a random solvable puzzle
    Print the initial board configuration
    Solve the puzzle and obtain the solution details
    If a solution is found:
        print the solution details: steps, nodes explored, max queue size, and time taken
    Else:
        print "No solution found"
```

Here is the Python code for Greedy Best First Search

```
import heapq
import random
import time

class PuzzleState:
    def __init__(self, tiles, empty_position, steps=0, previous=None):
        self.tiles = tiles
        self.empty_position = empty_position
        self.steps = steps
        self.previous = previous
        self.heuristic = self.calculate_manhattan_distance()

    def calculate_manhattan_distance(self):
        distance = 0
        for row in range(3):
            for col in range(3):
                value = self.tiles[row][col]
                if value != 0:
                    goal_row = (value - 1) // 3
                    goal_col = (value - 1) % 3
                    distance += abs(goal_row - row) + abs(goal_col - col)
        return distance

    def find_possible_moves(self):
        moves = []
        x, y = self.empty_position
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_tiles = [row[:] for row in self.tiles]
                new_tiles[x][y], new_tiles[new_x][new_y] = new_tiles[new_x][new_y], new_tiles[x][y]
                moves.append(PuzzleState(new_tiles, (new_x, new_y), self.steps + 1, self))
        return moves

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

    def is_goal_state(self):
        return self.tiles == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    def print_board(self):
        for row in self.tiles:
            print(" ".join(str(tile) if tile != 0 else " " for tile in row))
        print()

def count_inversions(tiles):
    flat_tiles = [num for row in tiles for num in row if num != 0]
    inversions = 0
    for i in range(len(flat_tiles)):
        for j in range(i + 1, len(flat_tiles)):
            if flat_tiles[i] > flat_tiles[j]:
                inversions += 1
    return inversions

def check_solvable(tiles):
    return count_inversions(tiles) % 2 == 0
```

Greedy Best First Search Algorithm continued

```
def create_random_puzzle():
    numbers = list(range(9))
    random.shuffle(numbers)
    board = [numbers[i:i+3] for i in range(0, 9, 3)]
    while not check_solvable(board):
        random.shuffle(numbers)
        board = [numbers[i:i+3] for i in range(0, 9, 3)]
    empty_position = next((i, row.index(0)) for i, row in enumerate(board) if 0 in row)
    return board, empty_position

def solve_puzzle(start_board):
    empty_position = None
    for row in range(3):
        for col in range(3):
            if start_board[row][col] == 0:
                empty_position = (row, col)
                break
    initial_state = PuzzleState(start_board, empty_position)
    priority_queue = []
    heapq.heappush(priority_queue, initial_state)
    visited_states = set()
    visited_states.add(tuple(tuple(row) for row in start_board))
    move_counter = 0
    nodes_explored = 0
    max_queue_size = 0
    start_time = time.time()
    solution_path = []
    while priority_queue:
        max_queue_size = max(max_queue_size, len(priority_queue))
        current_state = heapq.heappop(priority_queue)
        nodes_explored += 1
        if current_state.is_goal_state():
            while current_state:
                solution_path.append(current_state)
                current_state = current_state.previous
            solution_path.reverse()
            break
        for move in current_state.find_possible_moves():
            state_tuple = tuple(tuple(row) for row in move.tiles)
            if state_tuple not in visited_states:
                visited_states.add(state_tuple)
                heapq.heappush(priority_queue, move)
            end_time = time.time()
            if solution_path:
                print("Solution Path:")
                for state in solution_path:
                    print(f"Move {move_counter}:")
                    state.print_board()
                    move_counter += 1
                return {
                    "steps": len(solution_path) - 1,
                    "nodes_explored": nodes_explored,
                    "max_queue_size": max_queue_size,
                    "time_taken": end_time - start_time
                }
            return None

        random_board, empty_position = create_random_puzzle()
        print("Random Initial Board:")
        for row in random_board:
            print(row)

        solution = solve_puzzle(random_board)
        if solution:
            print(f"\nSteps to solve: {solution['steps']}")
            print(f"Nodes explored: {solution['nodes_explored']}")
            print(f"Max queue size: {solution['max_queue_size']}")
            print(f"Time taken: {solution['time_taken']} seconds")
        else:
            print("No solution found.")
```

And finally Our Champion A* Search Algorithm

This Algorithm works by first creating a random puzzle that can actually be solved. To do this, it shuffles the pieces and then checks if the puzzle is solvable by counting how many pairs of pieces are out of order. If the number of these "inversions" is even, the puzzle then can be solved but if it's odd, then it cannot. The program keeps shuffling the pieces up until it finds a solvable puzzle.

Once the puzzle is ready, then the algorithm starts trying to solve it using a strategy called A* search. This strategy works like playing the game by always making the move that seems to bring the puzzle closer to being solved. It does this by calculating something called the "Manhattan distance," which measures how far each tile is from where it should be. The program then explores all the possible moves it can make from the current puzzle layout and chooses the best one, based on which move reduces this distance the most. As the algorithm tries different moves, it keeps track of the ones that have been tried already, so it doesn't get stuck in a loop doing the same things over and over endlessly. It continues this process, exploring different possibilities, until it either solves the puzzle or runs out of options.

If the puzzle is solved, then the algorithm will show you the steps it took to land in that conclusion, how many moves were made, how many different puzzle layouts it explored, and how long the whole process took. If it cannot find a solution, which is very rare because the puzzle is solvable by design, it will tell you. The algorithm combines a bit of randomness with a smart search strategy to efficiently solve the puzzle.

Let Us Introduce it by Showing the A * algorithm Pseudocode First.

```
Node:
 Initialize(data, level, fval):
  self.data = data
  self.level = level
  self.fval = fval
 GenerateChild():
  (x, y) = Find(self.data, '0')
  val_list = [(x, y-1), (x, y+1), (x-1, y),
(x+1, y)]
  children = []
  For each (x2, y2) in val_list:
   child = Shuffle (self.data, x, y, x2,
y2)
   If child is not None:
    child_node = Node(child, self.level
+ 1, 0)
    children.append(child_node)
  Return children
 Shuffle(puz, x1, y1, x2, y2):
  If x2 and y2 are within bounds of
puz:
   temp_puz = Copy(puz)
   temp = temp_puz[x2][y2]
   temp_puz[x2][y2] =
temp_puz[x1][y1]
   temp_puz[x1][y1] = temp
   Return temp_puz
  Else:
   Return None
 Copy(root):
  Return deep copy of root
 Find(puz, x):
  For each (i, j) in puz:
   If puz[i][j] == x:
    Return (i, j)
  Return (-1, -1)
Puzzle:
 Initialize(size):
  self.n = size
  self.open = []
  self.closed = []

 ReadValue():
  While True:
   Try:
    puz = []
    specialcharcount = 0
    For _ in range(self.n):
     row = input("Enter row {_+1} (e.g.,
'1 2 3'):")
     row_list = row.split()
     If length of row_list != self.n or
contains invalid characters:
      Raise ValueError
     specialcharcount += count of '0'
in row_list
     puz.append(row_list)
    If specialcharcount != 1:
     Raise ValueError
    Return puz
   Except ValueError:
    Print error message


 CalFVal(start, goal):
```

```
  Return CalHVal(start.data, goal) +
start.level
                                        self.open.sort by fval in ascending
                                    order


 CalHVal(start, goal):
  temp = 0
  For each (i, j) in start:
   If start[i][j] != goal[i][j] and start[i][j]
!= '0':
    temp += 1
  Return temp
 Process():
  Print instructions
  start = ReadValue()
  goal = ReadValue()
  Print start and goal states
  start_node = Node(start, 0, 0)
  start_node.fval =
CalFVal(start_node, goal)
  self.open.append(start_node)
  While self.open is not empty:
   cur = self.open[0]
   Print cur.data, cur.level, cur.fval
   If CalHVal(cur.data, goal) == 0:
    Print solution found
    Break
   For each child in
cur.GenerateChild():
    If child not in self.closed:
     child.fval = CalFVal(child, goal)
     self.open.append(child)
   self.closed.append(cur)
   self.open.pop(0)
```

Lets Now Introduce the Actual Python Code

```
class Node:
    def __init__(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval
# generates possible child moves based on current position
    def generate_child(self):
        x, y = self.find(self.data, '0')
        val_list = [[x, y-1], [x, y+1], [x-1, y], [x+1, y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data, x, y, i[0], i[1])
            if child is not None:
                child_node = Node(child, self.level + 1, 0)
                children.append(child_node)
        return children
# shuffle the data position
    def shuffle(self, puz, x1, y1, x2, y2):
        if x2 >= 0 and x2 < len(puz) and y2 >= 0 and y2 < len(puz[0]):
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
            return temp_puz
        else:
            return None

    def copy(self, root):
        return [row[:] for row in root]

    def find(self, puz, x):
        for i in range(len(puz)):
            for j in range(len(puz[i])):
                if puz[i][j] == x:
                    return i, j
        return -1, -1


class Puzzle:
    def __init__(self, size):
        self.n = size
        self.open = []
        self.closed = []
```

```
#reads User input and handle input validations
    def readValue(self):
        while True:
            try:
                puz = []
                specialcharcount = 0
                for _ in range(self.n):
                    row = input(f"Enter row {_+1} (e.g., '1 2 3'):\n").strip()
                    row_list = row.split()
                    if len(row_list) != self.n:
                        raise ValueError("Row must contain exactly 3 elements.")
                    if not all(elem in {'1', '2', '3', '4', '5', '6', '7', '8', '0'
                        raise ValueError("Row contains invalid characters.")
                    specialcharcount += row_list.count('0')
                    puz.append(row_list)

                if specialcharcount != 1:
                    raise ValueError("Puzzle must contain exactly one '0' character
                return puz
            except ValueError as e:
                print(e)
# Calculates F Value -> f(n)=g(n)+h(n)
# g-depth of node
# h - no of misplaced tiles
    def calFVal(self, start, goal):
        return self.calHVal(start.data, goal) + start.level
# Calculates h value
    def calHVal(self, start, goal):
        temp = 0
        for i in range(self.n):
            for j in range(self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '0':
                    temp += 1
        return temp
```

```
# Calling readvalue method,getting user input for initial and goal states
# Generating child nodes, and their F values
# storing child nodes and sorting the list in ascending order
    def process(self):
        print("Value 0 denotes empty space in matrix for shuffle\n")
        print("Enter the initial state matrix of size 3x3\n")
        print("Example Input\n")
        print("1 2 3 \n 0 4 6 \n 7 5 8\n")
        start = self.readValue()
        print("Enter the goal state matrix of size 3x3\n")
        goal = self.readValue()
        print("Entered initial state\n")
        for row in start:
            print(" ".join(row))
        print("Entered final state\n")
        for row in goal:
            print(" ".join(row))

        start_node = Node(start, 0, 0)
        start_node.fval = self.calFVal(start_node, goal)

        self.open.append(start_node)
        print("\n ---------------------------** A star Algorithm for 8- Puzzle**--
        while self.open:
            cur = self.open[0]
            for row in cur.data:
                print(" ".join(row))
            print("Level ", cur.level)
            print("f(n) value ", cur.fval)
            if self.calHVal(cur.data, goal) == 0:
                print("Solution found at level ",cur.level)
                print (cur.data)
                break
```

```
            for child in cur.generate_child():
                if child not in self.closed:
                    child.fval = self.calFVal(child, goal)
                    self.open.append(child)
            self.closed.append(cur)
            self.open.pop(0)
            self.open.sort(key=lambda x: x.fval, reverse=False)

puz = Puzzle(3)
puz.process()
```

5. Input and Output

Our implementation takes an initial, scrambled state of the 8-puzzle as input and provides the sequence of moves needed to reach the goal state as output.

Sample Input

The input to our program is a scrambled 8-puzzle, represented as a list of numbers from 0 to 8, where 0 represents the empty space. For example:

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

This might look different based on the testing by one of our team members but corresponds to the following grid:

2 8 3
1 6 4
7 0 5

Sample Output

Our algorithms will take in this input and through various means they will manipulate the puzzle to reach the goal state. While the exact output of the algorithm may vary slightly, the output that is uniform across them is the acknowledgement that the proper solution has been found. In the instance of our DFS and BFS algorithm they simply notify the user that “A solution has been found” along with the number of nodes that were generated to achieve that result As for the Greedy Best Fit algorithm it returns the confirmation message, along with the number of nodes explored, the size of the queue, and the time taken to achieve the goal state.IDS outputs “solution found” and number of states explored along with the puzzle..Where as for the A* algorithm it includes the level it found the solution at along with the confirmation message and displays the process by which the goal state was reached.As this algorithm has our most detailed output we will provide an example below.

Actual Input:

```
Value 0 denotes empty space in matrix for shuffle

Enter the initial state matrix of size 3x3

Example Input

1 2 3
 0 4 6
 7 5 8

Enter the goal state matrix of size 3x3

Entered initial state

1 2 3
0 4 6
7 5 8
Entered final state

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

Actual Output:

```
----------------------------** A star Algorithm for 8- Puzzle**-------------------------------------

1 2 3
0 4 6
7 5 8
Level  0
f(n) value  3
1 2 3
4 0 6
7 5 8
Level  1
f(n) value  3
1 2 3
4 5 6
7 0 8
Level  2
f(n) value  3
1 2 3
4 5 6
7 8 0
Level  3
f(n) value  3
Solution found at level  3
[['1', '2', '3'], ['4', '5', '6'], ['7', '8', '0']]
```

6. Test Cases

We thoroughly tested our algorithms with various test cases to ensure their robustness and efficiency. Our test cases ranged from very simple to extremely challenging configurations in order to ensure its proper functioning.

Test Case Examples

1. **Easy Puzzle**:
    - **Input**: `[1, 2, 3, 4, 5, 6, 7, 0, 8]`
    - **Expected Output**: A few moves, such as moving '8' into the correct position.
2. **Medium Puzzle**:
    - **Input**: `[2, 8, 3, 1, 6, 4, 7, 0, 5]`
    - **Expected Output**: More complex series of moves to reach the goal state.
3. **Hard Puzzle**:
    - **Input**: `[5, 6, 7, 4, 0, 8, 3, 2, 1]`
    - **Expected Output**: A significant number of moves due to the high degree of scrambling.
4. **Edge Case** (Already Solved):
    - **Input**: `[1, 2, 3, 4, 5, 6, 7, 8, 0]`
    - **Expected Output**: No moves required.

7. Performance Analysis

We evaluated the performance of each algorithm based on several key metrics: speed, accuracy, and resource usage. Our goal was to understand how well each algorithm performs under different conditions and configurations of the puzzle.

Analysis Criteria We Used To Evaluate the performance.

1. **Speed**: Time taken to find the solution using several combinations of the puzzle.
2. **Accuracy**: Whether the algorithm finds the optimal solution (minimum moves before getting it done).
3. **Resource Usage**: Memory and computational power required during execution.

Summary of our Findings

- **BFS**:
  - **Speed**: Slow for large depths due to exhaustive nature.
  - **Accuracy**: Always finds the shortest path.
  - **Resource Usage**: High memory usage due to storing all nodes at each level.
- **DFS**:
  - **Speed**: Can be fast if a solution is found in early branches.
  - **Accuracy**: Not guaranteed to find the shortest path.
  - **Resource Usage**: Lower memory usage compared to BFS.
- **A***:
  - **Speed**: Generally the fastest due to heuristic guidance.
  - **Accuracy**: Often finds the optimal path quickly.
  - **Resource Usage**: Dependent on the heuristic, can be high.
- **Greedy Best-First**:
  - **Speed**: Fast due to heuristic guidance.
  - **Accuracy**: Not guaranteed to find the optimal path.
  - **Resource Usage**: Dependent on the heuristic.
- **Iterative Deepening Search**:
  - **Speed**: Moderate, combining benefits of BFS and DFS.
  - **Accuracy**: Finds the shortest path.
  - **Resource Usage**: More efficient than BFS in terms of memory.

8. Error Handling

Our implementation includes error handling to manage unexpected inputs and conditions, ensuring that the program runs smoothly without crashing.

Handled Errors

For the BFS Algorithm we implemented an input check when the user provides the puzzle to solve. It first checks to ensure every entry for the puzzle is a number between 0 and 8. It then ensures that there are 9 numbers provided so the puzzle will be full as well.

Very similar to the BFS, DFS checks for valid user inputs, as well as checks if each swap is a valid move within the puzzle.

In Greedy Best First Search, the algorithm handles errors by ensuring the puzzle is solvable before solving, using a set to avoid revisiting the same states and prevent infinite loops. It tracks memory usage by monitoring the maximum size of the priority queue and measures time taken to evaluate performance. Valid moves are ensured through the find_possible_moves method, and if the priority queue empties without finding a solution, the algorithm returns “none”, indicating no solution exists. However, since the puzzle is always solvable, this isn’t the case

For a* algorithm, we accept user inputs for initial and goal states. We have input validation entered values should be between 0 to 8 and input should have at least 1 0 value denoting as empty space.
Once the goal state is found, we will display a message to the user as the goal found. If not, the program will terminate if no goal is found.

The IDS with Error Handling algorithm has been written to be strong and resist most of the common issues, such as invalid start nodes or possibly missing neighbors within the graph. It ensures safe practices by checking the existence of the node and safe access methods to data for its reliable operation. While this broad exception handling covers all possible errors, more specific treatment in terms of exceptions and input validation could further try to improve the reliability of the code.

The whole implementation shows well how to walk a graph in BFS and DFS and, at the same time, captures the important error-handling mechanisms that provide stability and reliability of the program.

9. Optimization

Efficient Node Management:

The BFS algorithm in the 8-puzzle implementation utilizes a systematic approach to managing nodes and their children. By storing and exploring child nodes systematically and avoiding revisiting already explored nodes, the search process is made more efficient. The use of deep copies also ensures that the puzzle states are managed independently, preventing unintended modifications that could affect the search process.

Priority in Node Exploration:

While the implementations provided do not explicitly use priority queues, they optimize node exploration in BFS by systematically generating and exploring child nodes based on possible moves (Up, Down, Left, Right). This method ensures that all potential paths are explored without checking the same once again.

Depth and Breadth-First Traversal:

In the graph traversal implementation, both Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms are utilized. DFS optimizes space usage by exploring nodes deeply before backtracking, which is beneficial in scenarios with large but shallow search trees. BFS, on the other hand, guarantees the shortest path solution by exploring nodes level by level.

10. Assumptions and Limitations

We made certain assumptions during our implementation, and there are limitations to consider.

Assumptions

1. **Solvable Puzzles**: We assume that all input puzzles are solvable.
2. **Standard 3 * 3 Grid**: The solution is designed specifically for a standard 8-puzzle grid.

Limitations

1. **Computational Resources**: Some algorithms, especially BFS, require significant memory and processing power and our computer (mostly laptops) did not really help with fast response.
2. **Heuristic Quality**: The effectiveness of A* and Greedy Best-First is heavily dependent on the quality of the heuristic used.

11. Results and Conclusion

Our extensive testing and performance analysis led to the following conclusions:

Results

- **A* Search** provided the best balance of speed and accuracy, consistently finding the optimal solution quickly.
- **BFS** was thorough and accurate but slow and memory-intensive.
- **DFS** was fast but not always optimal.
- **Greedy Best-First** was fast but sometimes missed the optimal path.
- **Iterative Deepening Search** combined the strengths of BFS and DFS, performing well in both speed and accuracy.

Conclusion

The A* Search Algorithm emerged as the most effective method for solving the 8-puzzle problem, primarily due to its use of heuristics which guide the search efficiently towards the goal state. These heuristics provide more accurate estimates of the cost to reach the goal state, making the search process for us more efficient. However, the effectiveness of A* and other heuristic-based algorithms are heavily dependent on the quality of the heuristic used. It is obvious that Poor heuristics can lead us to underperformance and increased computational costs in the long run or in bigger projects than this one.

While BFS guarantees the shortest path, its high memory usage makes it impractical for larger or more complex puzzles. DFS, with its lower memory requirements, can be useful in scenarios where memory is not as abundant, but its tendency to explore deep branches without guarantee of finding the shortest path limits its reliability for larger data. Greedy Best-First Search offers speed but at the cost of completeness and optimality (based on the chart we provided), making it suitable for scenarios where a quick solution is preferred over the best solution but might not be the right answer. Iterative Deepening Search provides a balance between BFS and DFS, providing a good compromise in terms of memory usage and path optimality.

Future improvements could focus on sharpening the heuristics used in the A* algorithm and exploring optimization techniques to further enhancements. Overall, the A* Search Algorithm, complemented by well-chosen heuristics can provide a robust and efficient solution for the 8-puzzle problem, demonstrating the power of informed search techniques in artificial intelligence.