AN INTELLIGENT PUZZLE SOLVER

ABOUT PROJECT

Project Idea:

The project aims to develop an intelligent solver for the N-puzzle using a best-first search algorithm. These puzzles are classic sliding puzzles where the objective is to arrange numbered tiles in a specific order by sliding them into an empty space. The solver will utilize a heuristic-based approach to efficiently find the optimal solution, making it a useful tool for educational purposes, research, and demonstrating advanced search algorithms in artificial intelligence.

Overview:

The N-puzzle solver project is designed to tackle three popular sliding puzzles: the 8-puzzle (3x3 grid) , the 15-puzzle (4x4 grid), and the 25-puzzle (5x5 grid). Each puzzle presents a different level of complexity and challenge, providing a comprehensive test-bed for the algorithm. The best-first search algorithm is chosen for its efficiency in exploring the most promising paths first, guided by a heuristic function that estimates the cost to reach the goal state. The solver will be implemented in Python, leveraging data structures such as priority queues to manage the search process.

The project will be divided into the following stages:

1. Research and Planning: Understanding the puzzles, algorithm, and heuristic functions.
2. Implementation: Coding the solver, including the heuristic function, priority queue, and state manipulation.
3. Testing and Validation: Testing the solver on various puzzle instances to ensure correctness and efficiency.
4. Documentation: Creating comprehensive documentation for users and developers, including usage instructions and example runs.

Expected Outcomes:

By the end of the project, we aim to have a robust, efficient, and well-documented puzzle solver that can handle (8, 15, 25)-puzzle instances, providing valuable insights into heuristic search algorithms and their applications. And it'll have an easy-to-use design so people can play without any problem and have a nice time

PROJECT COMPONENTS

1. Best-First Search Algorithm

the algorithm uses a heuristic to estimate the cost to reach the goal from each node and expands the node with the lowest estimated cost first.

2.Heuristic Function:

The solver will implement various heuristic functions (Manhattan distance, misplaced tiles, Euclidean,Linear conflict) to guide the search process.

3.State Representation:

Efficient representation and manipulation of puzzle states to facilitate quick computation and comparison.

4.Solution Path Reconstruction:

Once the goal state is reached, the solver will reconstruct the path of moves leading to the solution.

5.User Interface:

The user-friendly Interfaces provide an intuitive layout with clear navigation and attractive visuals. And enhancing the overall experiment

Similar Applications

1. Travelling Salesman Problem (TSP)

- Description: Finding the shortest possible route that visits a set of cities exactly once and returns to the origin city.
- Application: Important in operations research and demonstrates optimization algorithms like genetic algorithms and simulated annealing.

2. Rubik's Cube

- Description: A 3D combination puzzle where the objective is to arrange the cube so that each face is a single solid color.
- Application: Demonstrates advanced problem-solving techniques and heuristic search algorithms in a 3D space.

3. Sokoban

- Description: A puzzle game where the player pushes boxes around a warehouse, trying to get them to designated storage locations.
- Application: Useful for studying pathfinding and planning algorithms, as it involves constraints and requires careful planning of moves.

4.Knight's Tour

- Description: A chess problem where the knight must visit every square on the chessboard exactly once.
- Application: Used in studying backtracking and pathfinding algorithms

A LITERATURE REVIEW OF ACADEMIC PUBLICATIONS

1. Solving Mathematical Puzzles: A Challenging Competition for AI By Federico Chesani ,Paola Mello and Michela Milan

   URL:https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/2736

2. Puzzle Solving without Search or Human Knowledge: An Unnatural Language Approach By David Noever, Ryerson Burdick

   URL:https://arxiv.org/abs/2109.02797

3. Finding optimal solutions to the twenty-four puzzle By RE Korf, LA Taylor

   URL:https://citeseerx.ist.psu.edu/document?repid=rep1&tye=pdf&doi=a3b936d9302d793a9438b3d2adbdd1924fb0d42a

4. Artificial intelligence and problem solving By D Kopec, C Pileggi, D Ungar, S Shetty (Book)

   URL:https://books.google.com/books?hl=en&lr=&id=UAyDgAAQBAJ&oi=fnd&pg=PP13&dq=intelligent+puzzle+solver&ots=SXKZ1Jwk7u&sig=sUBzb5xkPnuSWqeAlydI4TzB9o8

5. Puzzle-based learning: An introduction to critical thinking and problem solving ByZ Michalewicz, N Falkner, R Sooriamurthi

   URL:https://www.andrew.cmu.edu/user/sraja/papers/2011decsionline-oct-paper.pdf

Main functionalities

The intelligent (8, 15, 25)-puzzle solver is designed to efficiently find solutions to sliding tile puzzles using a best-first search algorithm. The solver's main functionalities include:

1. Initialization and Input Handling

- Reading Puzzle Input: Accepts the initial puzzle state as input, either through a command-line interface or a graphical user interface.
- State Validation: Ensures the input state is valid and solvable, checking for the correct number of tiles and the presence of exactly one empty space.

2. Select one Heuristic Functions from:

- Manhattan Distance: Calculates the sum of the absolute differences between the current and goal positions of each tile.
- Misplaced Tiles: Counts the number of tiles that are not in their goal positions.
- Euclidean Distance: Computes the straight-line distance between the current and goal positions of each tile.
- Linear Conflict: Enhances the Manhattan Distance by adding penalties for tiles that are in the correct row or column but in the wrong order.

3.Output and Visualization

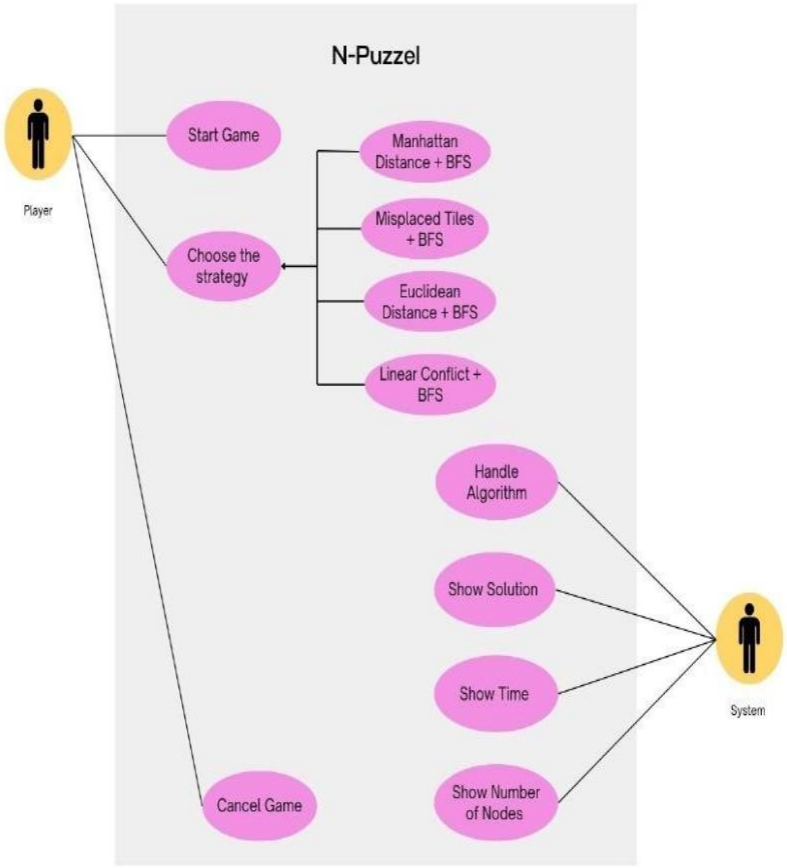
- Solution Display: Outputs the sequence of moves required to solve the puzzle.
- Intermediate Steps: Optionally displays intermediate states to show the progress of the solution.
- Metrics Reporting: Provides metrics such as the number of moves, the time taken to find the solution, and the number of nodes expanded.

4.User Interface

- Command-Line Interface: Allows users to input the initial state and select the heuristic function to be used.
- Graphical User Interface (Optional): Provides a visual representation of the puzzle and allows users to interact with the solver more intuitively.

5. Customization and Extension

- Heuristic Selection: Allows users to choose from the four heuristic functions (Manhattan distance, misplaced tiles, Euclidean, linear conflict).
- Parameter Tuning: Provides options to adjust parameters and settings for the search algorithm to optimize performance.



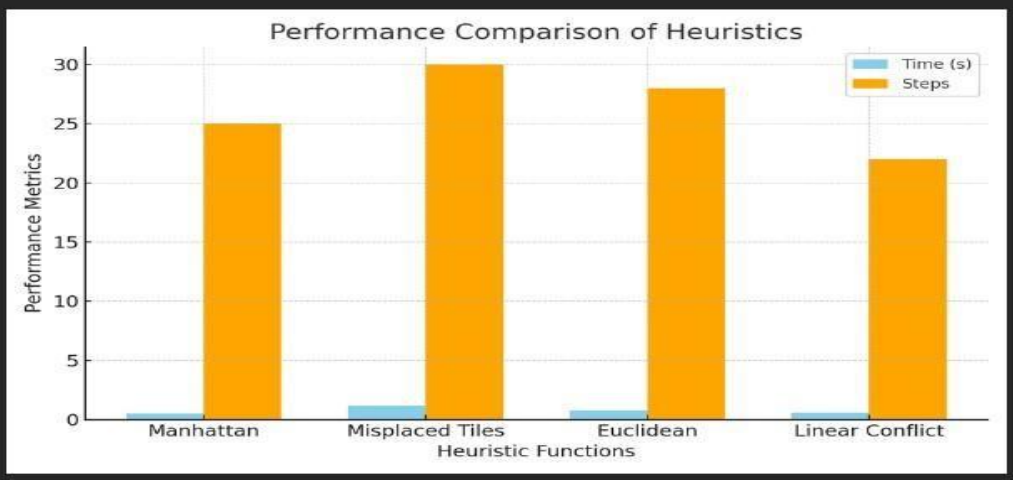
Experiments & Results:

Experiments: The project was tested with various puzzle sizes (3x3, 4x4, etc.) and different heuristic functions. The tests involved generating random solvable puzzles, applying the selected heuristic, and comparing the results in terms of the number of steps and time taken to solve the puzzle.

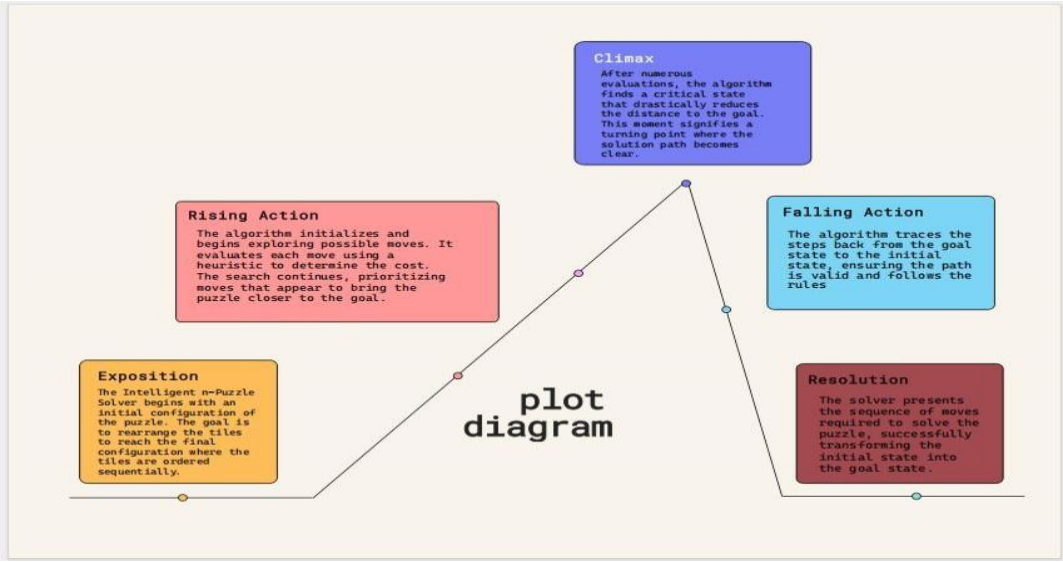
Results: The solver successfully solves the puzzles, and the results are displayed in terms of the number of steps taken and the time spent solving the puzzle. The animations demonstrate the puzzle being solved step by step.

5. Performance Plot

The plot below shows the comparison of heuristic functions in terms of time and steps:



Advantages And Disadvantages:



Advantages:

The program is flexible, allowing users to choose heuristics and observe their impact on the solution time and steps. Disadvantages:

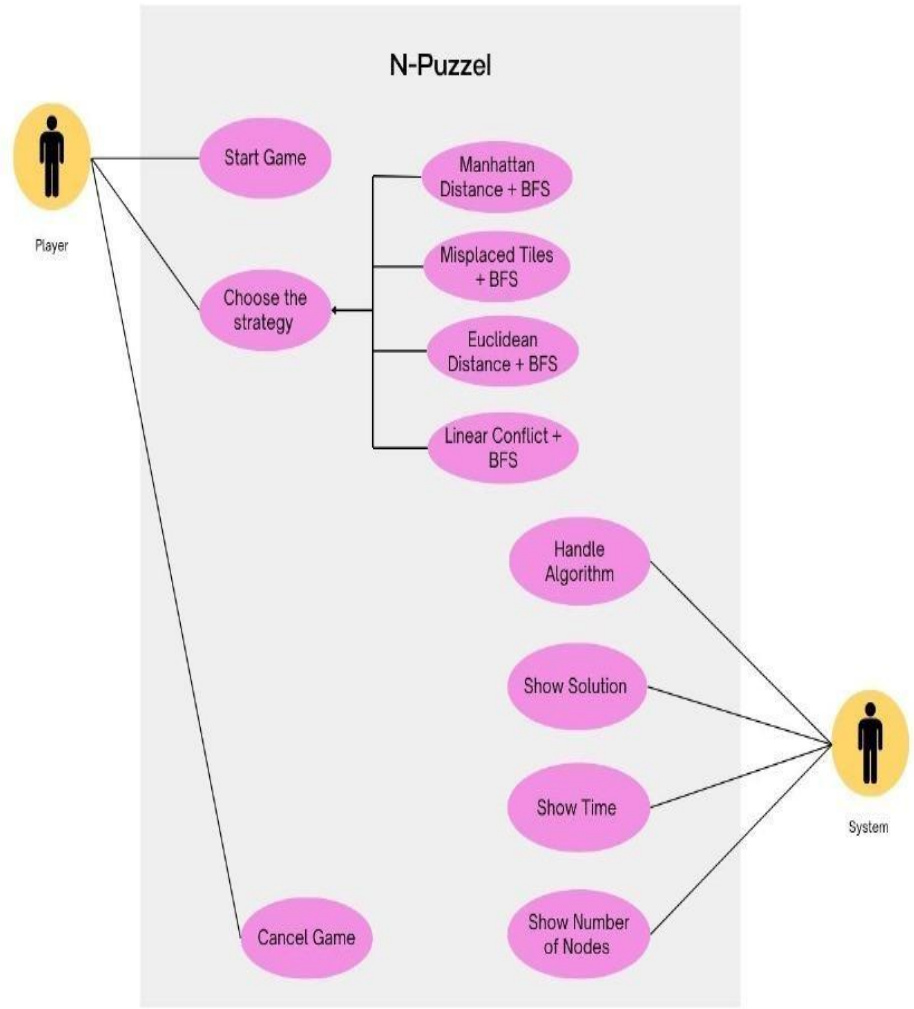
The algorithm may struggle with larger grids due to increased complexity, and some heuristics (like Linear Conflict) are computationally expensive.

Algorithm Behavior:

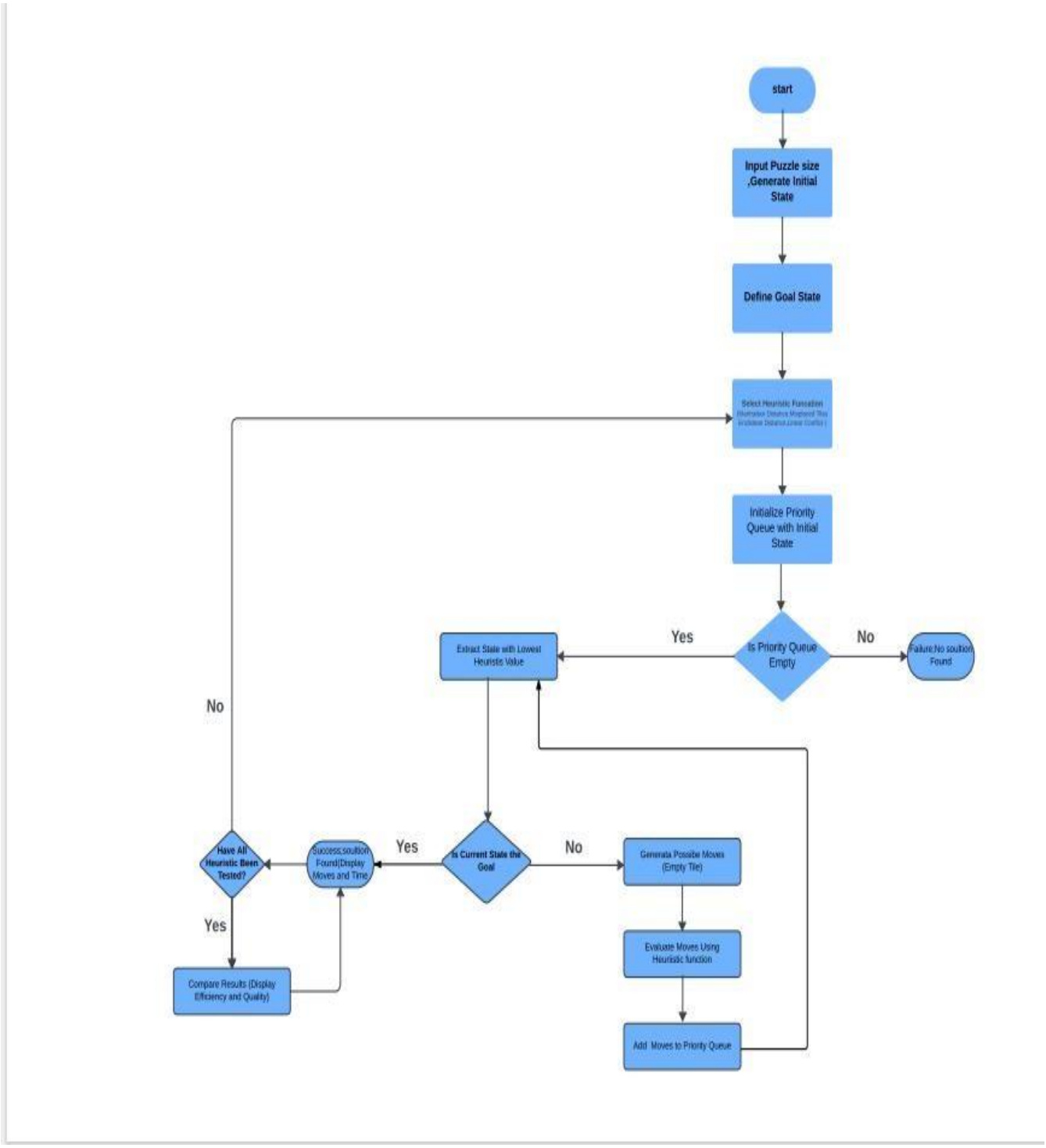
The algorithm's performance varies depending on the heuristic used. Manhattan Distance tends to be faster due to its simplicity, while Linear Conflict provides a more accurate estimation but at the cost of increased computation. Future modifications could involve optimizing the heuristic calculations for larger grids or implementing more advanced search (A*)

Diagrams:

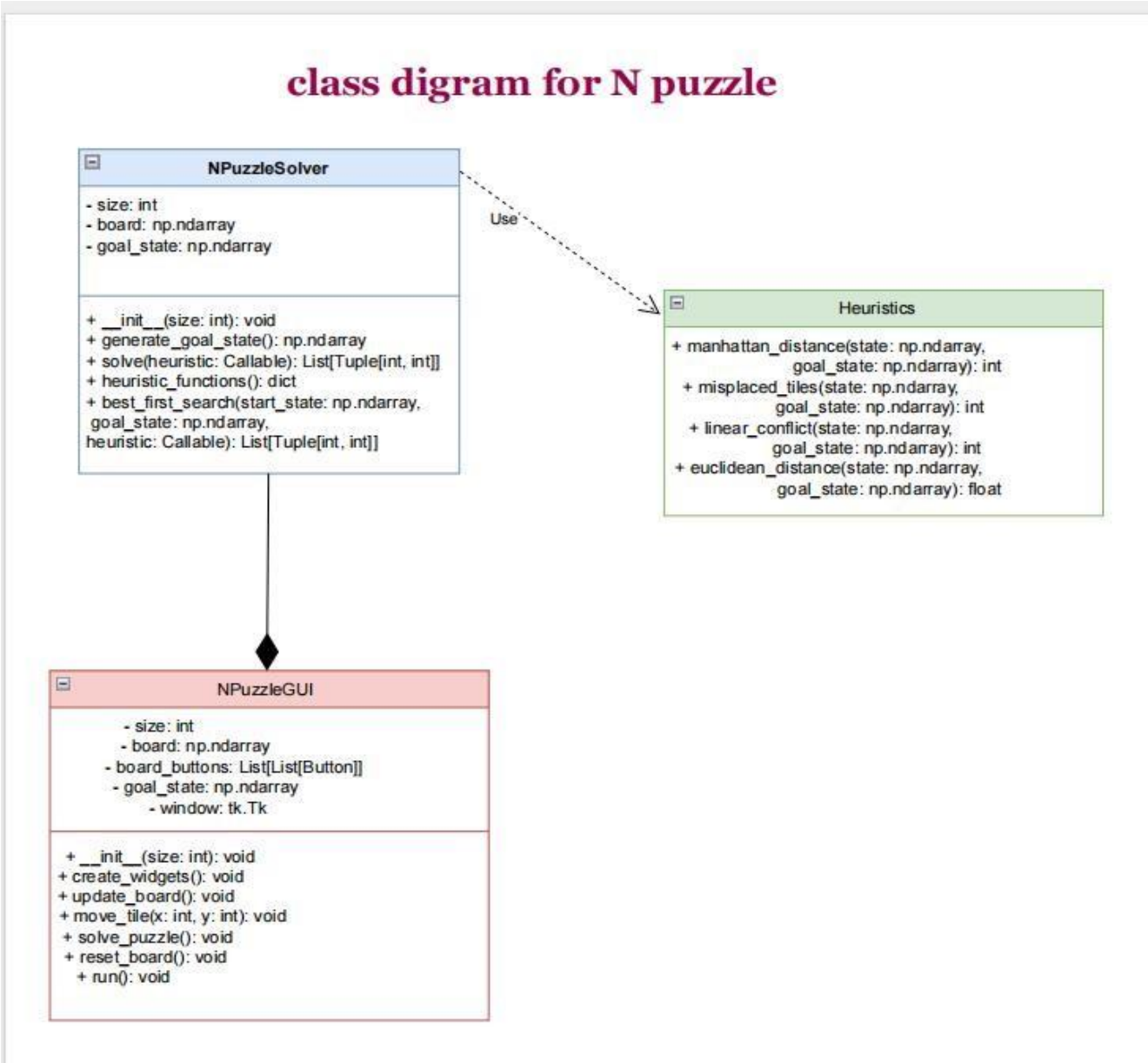
1.Use case


N-Puzzel
Player
Start Game
Choose the strategy
Manhattan Distance + BFS
Misplaced Tiles + BFS
Euclidean Distance + BFS
Linear Conflict + BFS
Cancel Game
Handle Algorithm
Show Solution
Show Time
Show Number of Nodes
System

2.Flow Chart


start
Input Puzzle size ,Generate Initial State
Define Goal State
Initialize Priority Queue with Initial State
Is Priority Queue Empty
Yes
No
Failure:No soultion Found
Extract State with Lowest Heuristic Value
Is Current State the Goal
Yes
No
Success,soultion Found(Display Moves and Time)
Have All Heuristic Been Tested?
No
Yes
Compare Results (Display Efficiency and Quality)
Generate Possible Moves (Empty Tile)
Evaluate Moves Using Heuristic function
Add Moves to Priority Queue

3.Class


class digram for N puzzle
NPuzzleSolver
- size: int
- board: np.ndarray
- goal_state: np.ndarray
+ __init__(size: int): void
+ generate_goal_state(): np.ndarray
+ solve(heuristic: Callable): List[Tuple[int, int]]
+ heuristic_functions(): dict
+ best_first_search(start_state: np.ndarray,
goal_state: np.ndarray,
heuristic: Callable): List[Tuple[int, int]]
Use
Heuristics
+ manhattan_distance(state: np.ndarray,
goal_state: np.ndarray): int
+ misplaced_tiles(state: np.ndarray,
goal_state: np.ndarray): int
+ linear_conflict(state: np.ndarray,
goal_state: np.ndarray): int
+ euclidean_distance(state: np.ndarray,
goal_state: np.ndarray): float
NPuzzleGUI
- size: int
- board: np.ndarray
- board_buttons: List[List[Button]]
- goal_state: np.ndarray
- window: tk.Tk
+ __init__(size: int): void
+ create_widgets(): void
+ update_board(): void
+ move_tile(x: int, y: int): void
+ solve_puzzle(): void
+ reset_board(): void
+ run(): void

4.Block Digram