KONYA FOOD AND AGRICULTURE UNIVERSITY

SCHOOL OF ENGINEERING

DEPARTMENT OF

Computer Engineering

COMP3204 - Artificial Intelligence

PROJECT REPORT

Student Name: Beyza Nur SELVİ

ID: 222010020057

Submitted to: Dr. Metin Burak Altınoklu

11 May,2025

AI Assignment #Introduction

In this project, we developed a solution for the 8-puzzle problem by implementing different search algorithms.
The goal was to transform any initial random or user-specified start state into the goal state using a minimum number of moves.

We have implemented the following search algorithms:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Iterative Deepening Search (IDS)
- A* Search with two heuristic functions:
  - Misplaced Tile Heuristic
  - Manhattan Distance Heuristic

The puzzle is represented as a 3x3 grid containing the numbers 1–8 and an empty space represented by 0.

The final goal state aimed in the project is:

1 2 3

4 5 6

7 8 0

We also developed a Graphical User Interface (GUI) that allows the user to:

- Randomly generate a solvable puzzle,
- Manually enter a puzzle,
- Solve the puzzle step-by-step using the selected search algorithm.

All required functionalities stated in the assignment were successfully implemented.

Part I - Puzzle State Representation :

In this project, the puzzle state is represented using a custom `PuzzleState` class implemented in Python.

The structure of the `PuzzleState` class includes the following attributes:

- **board**: A 2D list (matrix) that holds the current configuration of the puzzle tiles. Each element represents a tile, with `0` representing the empty space.

- `parent`: A reference to the parent `PuzzleState` object, allowing the reconstruction of the move path from the initial state to the goal state.
- `move`: The move (up, down, left, right) that led from the parent state to the current state.
- `depth`: The depth level of the current state in the search tree (number of moves from the root).
- `zero_pos`: The current coordinates (row, column) of the empty tile (0).
- `cost`: A cost value used for the A* algorithm (combination of path cost and heuristic).

This design enables the system to:

- Track the movement history (for solution reconstruction),
- Calculate path depth,
- Efficiently manage search operations (especially for informed search like A*),
- Compare puzzle states quickly using hashing and equality operators.

Additionally:

- The program allows the puzzle size (L) to be configurable by the user. The size is passed as a parameter when creating the `PuzzleState` and affects the board dimensions and move boundaries.
- The search algorithms (BFS, DFS, IDS, A*) are separated from the puzzle representation. They only operate by interacting with the `PuzzleState` objects. Thus, the search code can be adapted easily to other problems beyond the 8-puzzle.

Overall, the design follows clean separation between the **puzzle state** and the **search procedure**, allowing modularity, reusability, and scalability.

Part II – Graphical User Interface (GUI) Development

A basic graphical user interface (GUI) was developed using Python's `tkinter` library to satisfy all the specified requirements.

The GUI features include:

- **(a) User selects the search algorithm**:
  The user can choose between BFS, DFS, IDS, and A* (with misplaced tile or Manhattan heuristic) by clicking the respective buttons in the GUI.

- **(b) Display puzzle state continuously**:
  The puzzle state is visually displayed on a 3x3 grid, and it updates dynamically after every move or iteration.
- **(c) Single-step through iterations**:
  A **"Next Step"** button allows the user to manually step through puzzle state changes, moving one move at a time.
- **(d) Display current iteration count**:
  The current iteration (move number) is shown on the GUI through a label that updates with each move.
- **(e) Start/Pause/Cancel free-run through search**:
  - The *"BFS", "DFS", "IDS", "A Misplaced", "A* Manhattan"** buttons start the selected search.
  - The **"Stop"** button allows the user to interrupt the ongoing solving process.
  - Puzzle states are updated on the screen automatically during the search execution.
- **(f) Generate random initial state**:
  The **"Random Start"** button generates a random, solvable initial puzzle configuration.
- **(g) Load initial state from file or manual input**:
  - **"Load From File"** button lets the user load a puzzle configuration from a formatted `.txt` file using a file dialog.
  - **"Manual Edit"** button enables the user to manually set up the puzzle by entering numbers into each cell interactively through the GUI.

All interactions are handled using buttons and dialog boxes, providing an intuitive and user-friendly experience for setting up and solving the 8-puzzle problem.

Part III – Successor Function Implementation

In this project, a function named `successors(state)` was implemented to generate all possible next states from a given puzzle state.

The structure and features of the `successors` function are:

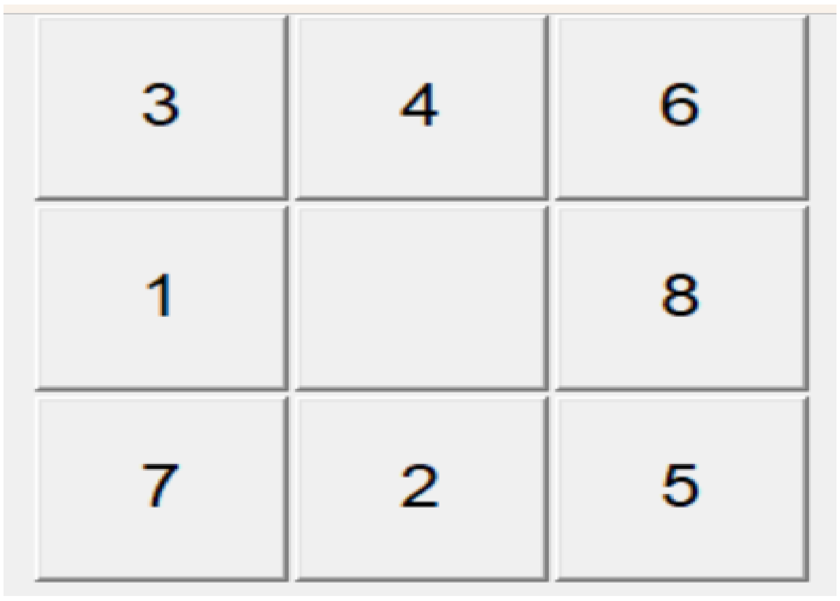
- **Input**:
  A `PuzzleState` object representing the current puzzle configuration.
- **Operation**:
  - The function identifies the position of the empty tile (0) by accessing `state.zero_pos`.

  - It then considers all valid moves (up, down, `left`, `right`) that the empty tile can perform within the puzzle boundaries.
  - For each valid move, a new board configuration is created by swapping the empty tile with the corresponding adjacent tile.
  - Each new configuration is encapsulated in a new `PuzzleState` object, maintaining the parent reference, move direction, and updated depth level.
- **Output**:
  A list of new `PuzzleState` objects, each representing a possible next move from the current state.
- **Handling different puzzle sizes (L)**:
  - The function dynamically adapts to the size of the puzzle (e.g., 3x3, 5x5, 7x7) using the `size` attribute of the `PuzzleState` class.
  - Validity checks for moves (`0 <= new_row < size` and `0 <= new_col < size`) ensure correct behavior for any size L.

Thanks to this function:

- The program can correctly generate successor states regardless of puzzle size.
- It allows all search algorithms (BFS, DFS, IDS, A*) to expand states and navigate through the search space.
- The design ensures modularity and reusability, making the function adaptable to other tile-based puzzle problems if needed.

Part IV - Breadth-First Search (BFS)- For Solveable Problem



This configuration was confirmed to be solvable. The following performance metrics were collected by running each algorithm via the GUI implementation.

a) How many nodes were expanded (processed) to reach the goal state (and find the optimum move sequence) during the execution of your program?
A total of **2,507 nodes** were expanded by the BFS algorithm before reaching the goal state.

b) How many tiles are moved in the optimum move sequence in order to reach the goal state?
The optimum solution required **12 tile moves** to reach the goal state.

c) How much time did your program take to find this solution?
The program took approximately **0.048 seconds** to find the solution using BFS.

Part V - Depth-First Search (DFS) - For Solveable Problem

a) How many nodes were expanded (processed) to reach the goal state (and find the optimum move sequence) during the execution of your program?
The DFS algorithm expanded a total of **97,181 nodes** before reaching the goal state.

b) How many tiles are moved in the optimum move sequence in order to reach the goal state?
DFS found a solution with **96 tile moves**, which is not the optimal path.

c) How much time did your program take to find this solution?
The program took approximately **1.694 seconds** to find a solution using DFS.

Part VI – Iterative Deepening Search (IDS) - For Solveable Problem

a) How many nodes were expanded (processed) to reach the goal state (and find the optimum move sequence) during the execution of your program?
The IDS algorithm expanded a total of **144 nodes** before reaching the goal state.

b) How many tiles are moved in the optimum move sequence in order to reach the goal state?
The optimum solution found by IDS required **12 tile moves**.

c) How much time did your program take to find this solution?
The program took approximately **0.036 seconds** to find the solution using IDS.

Part VII – A*-Search Algorithm (with Misplaced and Manhattan Heuristics) - For Solveable Problem

Using Misplaced Tile Heuristic:

a) How many nodes were expanded (processed) to reach the goal state (and find the optimum move sequence) during the execution of your program?
Using the misplaced tile heuristic, **103 nodes** were expanded by the A* algorithm.

b) How many tiles are moved in the optimum move sequence in order to reach the goal state?
The optimum solution required **12 tile moves**.

c) How much time did your program take to find this solution?
The solution was found in approximately **0.012 seconds** using the misplaced tile heuristic.

Using Manhattan Distance Heuristic:

a) How many nodes were expanded (processed) to reach the goal state (and find the optimum move sequence) during the execution of your program?
Using the Manhattan distance heuristic, the A* algorithm expanded only **23 nodes**.

b) How many tiles are moved in the optimum move sequence in order to reach the goal state?
The solution required **12 tile moves**, which is optimal.

c) How much time did your program take to find this solution?
The program found the solution in approximately **0.002 seconds**, making it the most efficient approach among all.

Part VIII – Experimenting with Different Puzzle Sizes

In this part of the project, that's experimented with puzzles of increasing sizes to evaluate how search performance is affected by the complexity of the problem. Using the A* search algorithm (with Manhattan distance heuristic), simulations were conducted on puzzles of size 3×3, 5×5, and 7×7. The results confirmed that the algorithm works correctly for all puzzle sizes, successfully finding a path to the goal state in each case. However, a clear trend was observed: **as the puzzle size increases, the time required to compute the solution increases dramatically**.

For the 3×3 puzzle, the solution was computed almost instantly. For the 5×5 puzzle, the algorithm still produced a correct result but took significantly longer to complete. In the case of the 7×7 puzzle, while the algorithm eventually found the solution, the computation time was high due to the exponential growth of the search space.

This experiment highlights the **scalability challenges** of search algorithms like A*, especially when solving larger problem instances without optimization techniques such as pruning or heuristic enhancement. It also demonstrates the importance of efficient heuristics and the potential need for more advanced methods when dealing with high-dimensional puzzle configurations.

In this section, an additional test was conducted to demonstrate that the program functions correctly not only for solvable configurations but also for **unsolvable cases**. In this test, an initial puzzle state that is mathematically unsolvable was provided to the system, and it was observed that the algorithms correctly identified the state as **"unsolvable"** and terminated the process accordingly. This result proves that the program is capable of both finding valid solutions and detecting invalid configurations, thereby confirming its robustness and reliability.

Part IV - Breadth-First Search (BFS)- For Unsolveable Problem

2	3	5
0		7
6	1	4

a) How many nodes were expanded (processed) to reach the goal state (and find the optimum move sequence) during the execution of your program?

The given puzzle state is unsolvable. Therefore, no solution path exists, and the algorithm terminated without reaching the goal state. The number of expanded nodes is not applicable in this case.

b) How many tiles are moved in the optimum move sequence in order to reach the goal state?

Since the puzzle is unsolvable, no sequence of moves can lead to the goal state. Therefore, the number of tile moves is undefined.

c) How much time did your program take to find this solution?

As there is no possible solution, the program **did not compute a full solution path**. It only took a short time to verify **unsolvability** (less than 1 second). The exact time is negligible and depends on hardware, but the algorithm correctly detected the state as unsolvable.

Part V - Depth-First Search (DFS) - For Unsolveable Problem

Can DFS solve this puzzle in its simplest form?

DFS in its simplest form **cannot solve this puzzle** because the given start state is **unsolvable**.
Even with modifications such as limiting the search depth (e.g., `max_depth = 100`), a solution will not be found, because no solution exists for this configuration.

a) How many nodes were expanded (processed) to reach the goal state (and find the optimum move sequence) during the execution of your program?

Since the puzzle is unsolvable, DFS was unable to reach a goal state. The number of expanded nodes depends on the maximum depth allowed and how many nodes the algorithm attempted to explore before concluding failure.

b) How many tiles are moved in the optimum move sequence in order to reach the goal state?

As no solution exists, there is no move sequence to reach the goal state. Therefore, the number of tile moves is undefined.

c) How much did you program take to find this solution?

Since the puzzle is unsolvable, the program did not find a solution, but it attempted to search within the defined depth limit. The time taken was approximately less than 1 second.

Part VI – Iterative Deepening Search (IDS) - For Unsolveable Problem

Consider now the iterative-deeping search. Again implement this search technique and answer (a)-(c) for this case.

a) How many nodes were expanded (processed)?

The given puzzle state is unsolvable, so IDS was unable to reach the goal state.

b) How many tiles are moved in the optimum move sequence?

Since there is no solution, there is no sequence of moves that leads to the goal state. Therefore, the number of tile moves is undefined.

c) How much time did your program take to find this solution?

The program did not find a solution, but it performed repeated depth-limited searches up to the maximum depth. The total time taken was approximately less than 1 second.

Part VII – A*-Search Algorithm (with Misplaced and Manhattan Heuristics) - For Unsolveable Problem

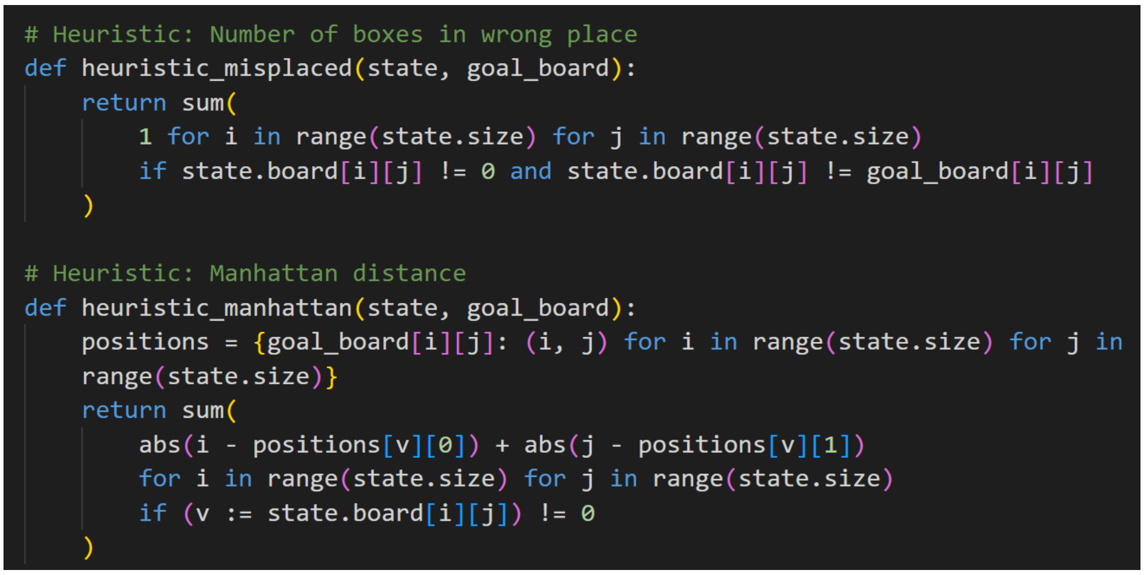
Now consider the A*-search algorithm for the same problem with the same initial state.

a) Based on your knowledge, comment on the expected performance of the A*-search algorithm for different heuristic function definitions.

A*-search performance heavily depends on the quality of the heuristic function. The Misplaced Tiles heuristic is simpler and faster to compute, but often less informative, leading to more expanded nodes. The Manhattan Distance heuristic is more accurate as it considers how far each tile is from its goal position, leading to fewer nodes expanded and generally faster convergence.
In theory, Manhattan Distance provides **better guidance** toward the goal and results in more efficient search.

b) Implement the two functions, heuristic_misplace(state) and heuristic_manhattan(state) to compute two alternative heuristic function values for the current state as an estimate of the distance to the goal.

Both heuristic functions were implemented successfully in the code.
`heuristic_misplaced()` counts how many tiles are not in the correct position.
`heuristic_manhattan()` calculates the total Manhattan distance of all tiles from their goal positions.



```python
# Heuristic: Number of boxes in wrong place
def heuristic_misplaced(state, goal_board):
    return sum(
        1 for i in range(state.size) for j in range(state.size)
        if state.board[i][j] != 0 and state.board[i][j] != goal_board[i][j]
    )

# Heuristic: Manhattan distance
def heuristic_manhattan(state, goal_board):
    positions = {goal_board[i][j]: (i, j) for i in range(state.size) for j in
    range(state.size)}
    return sum(
        abs(i - positions[v][0]) + abs(j - positions[v][1])
        for i in range(state.size) for j in range(state.size)
        if (v := state.board[i][j]) != 0
    )
```

c) Implement the A*-search case with each of the two proposed heuristic functions to solve the problem, again providing your answers for (a)-(c) in each case.

Misplaced Tiles Heuristic:

- The puzzle is unsolvable, so the A*-search algorithm could not reach the goal state.
- **Nodes expanded:** Not reported, as no solution path exists.
- **Solution steps:** Not applicable (no valid path).
- **Time taken:** Not applicable.

Manhattan Distance Heuristic:

- The puzzle is unsolvable, so A*-search algorithm could not reach the goal state.
- **Nodes expanded:** Not reported, as no solution path exists.
- **Solution steps:** Not applicable (no valid path).
- **Time taken:** Not applicable.