UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO - IC/UFF
PÓS-GRADUAÇÃO EM COMPUTAÇÃO - PPGC

OWOOLA OLUWAGBEMILEKE BAMISE

DESENVOLVIMENTO DE JOGOS DIGITAIS: *BUILDING AN 8-PUZZLE GAME WITH A LOGIC IN UNITY (C#)**

RIO DE JANEIRO

2025

Activity carried out for the Desenvolvimento de Jogos Digitais, taught by Professor Dr. Esteban Walter Gonzalez Clua, during the Postgraduate Program in Computing (PPGC).

TABLE OF CONTENTS:

BUILDING AN 8-PUZZLE GAME WITH A LOGIC IN UNITY (C#)*

Building the 8-Puzzle game and integrate A* (A-star) Pathfinding to solve it.
This is divided into three sections.

1. First, I will focus on implementing the 8 puzzle game functionality, emphasizing the programming aspects over design elements.
2. Next, implementing our pathfinding algorithms, including A*, Dijkstra's, and Greedy Best-First Search.
3. Finally, I will apply these pathfinding algorithms to solve various configurations of the 8-Puzzle.

INTRODUCTION TO 8-PUZZLE

1. Introduces the 8-Puzzle game and its integration with A* pathfinding in Unity. It covers the problem's history, its complexity, and heuristic search methods like A*. It also explains the puzzle's state representation and neighbor construction for pathfinding, providing foundational knowledge for implementing and solving the 8-Puzzle.

2. Creating a new Unity 3D project named "8-Puzzle". It details setting up the main camera, importing assets, and configuring a tile prefab. Creating a puzzle board frame using cubes with wood textures and set up a basic UI with four buttons and two text fields for randomization, image cycling, resetting, and solving the puzzle using A*.

3. Covers the implementation of the Puzzle State class for the 8-puzzle game in C#. represents a unique puzzle state with an array of tile indices and includes constructors, equality checks, a hash code generator, and methods for finding the empty tile, swapping tiles, and calculating Manhattan cost. Additionally, it details creating neighbor indices for grid cells and generating neighboring puzzle states by moving the empty tile.

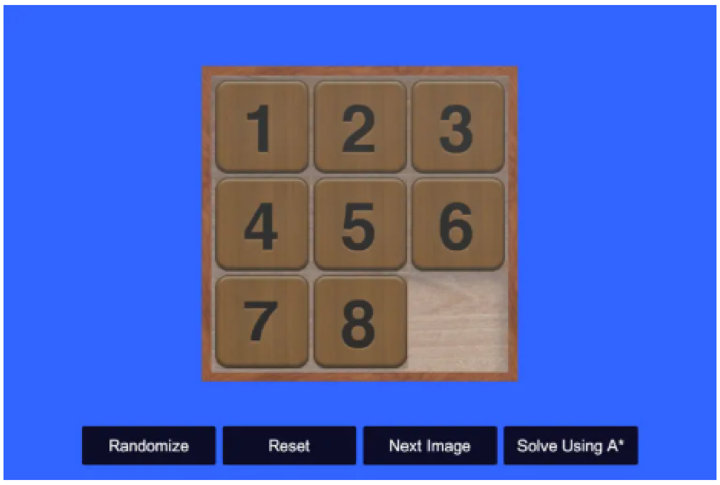
4. Implement the Path Finder using the three commonly used path finding algorithms, viz., the Dijkstra, the A* and the Gree-best-first search algorithms.

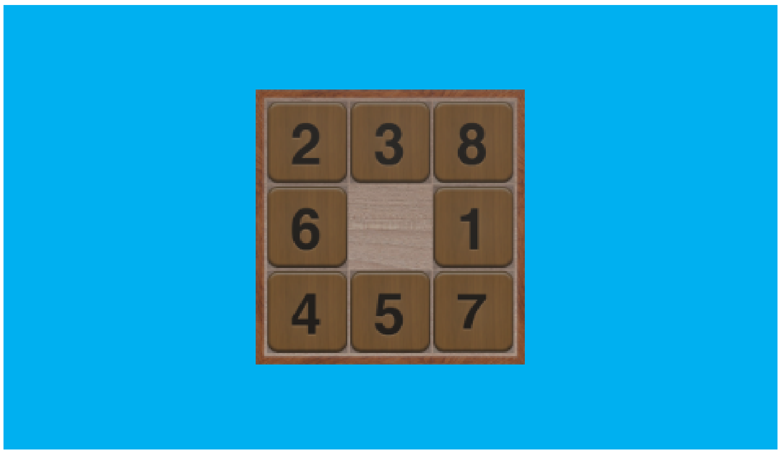
5. By applying the A* Path Finder to solve our 8-Puzzle game. Also create the necessary functionality to interactively play the game using the basic UI created.

IMPLEMENTED A GENERIC PATHFINDER IN UNITY USING C#

Implemented a generic pathfinder in Unity using C#. using the generic pathfinder and apply it to an 8-puzzle game. Demonstrate how our generic pathfinder not only works on a 2d grid-based system but also solves an 8-puzzle problem.

THE 8-PUZZLE PROBLEM

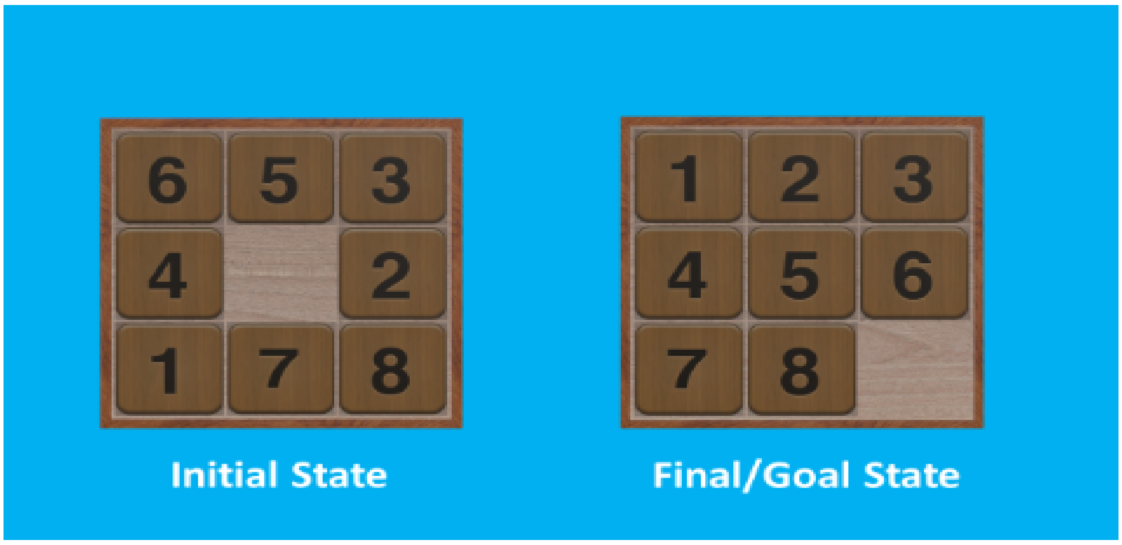




The 8-Puzzle problem was invented and popularised
by **Noyes Palmer Chapman** in the 1870s. It's a smaller version of the more well-known 15-puzzle, consisting of a

3-by-3 grid with eight numbered square blocks and one blank square.

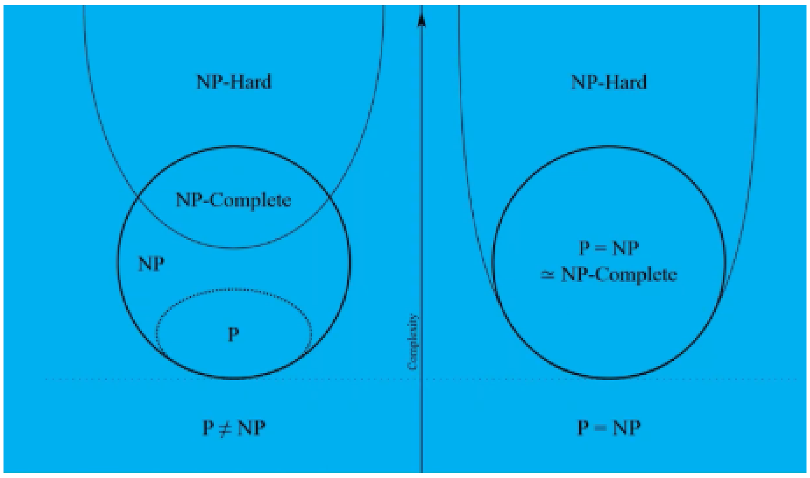
The objective of the puzzle is to rearrange the blocks into a specific order, typically shown with the blank square either at the beginning or end of the sequence.

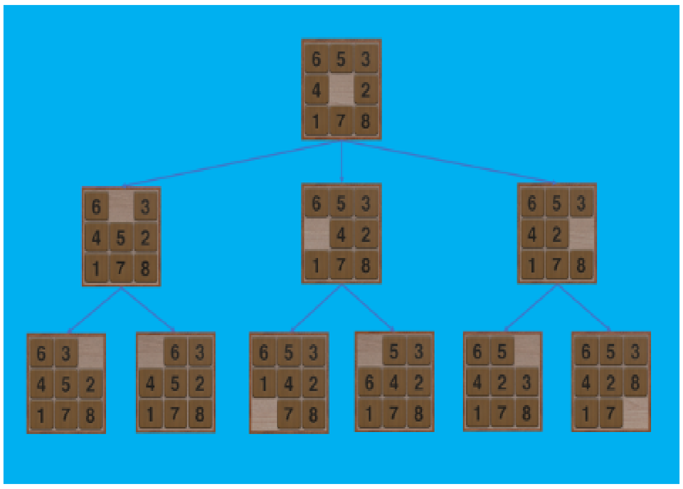


Blocks can be moved horizontally or vertically into the blank square to achieve the goal state.

The A*path-finding algorithm is commonly applied to grid-based pathfinding scenarios. However, in general, any pathfinding algorithm, including A*, can be utilised to solve graph-based problems.

The 8-Puzzle Solution Search Space
The 8-Puzzle represents the largest possible N-puzzle that can be completely solved. While it is straightforward, it presents a substantial problem space. Larger variants like the 15-puzzle exist but cannot be entirely solved.





HEURISTIC SEARCH:

Heuristic search is a method used to solve search problems more quickly than traditional methods. It often provides an approximate solution when conventional methods cannot, offering a generalised and approximate approach to problem-solving. In simple terms, heuristic search can be likened to a rule of thumb or common-sense knowledge. While the answer isn't guaranteed to be accurate, it aids in reaching a decision swiftly, sacrificing optimality, completeness, accuracy, or precision for speed.

THE A* SEARCH

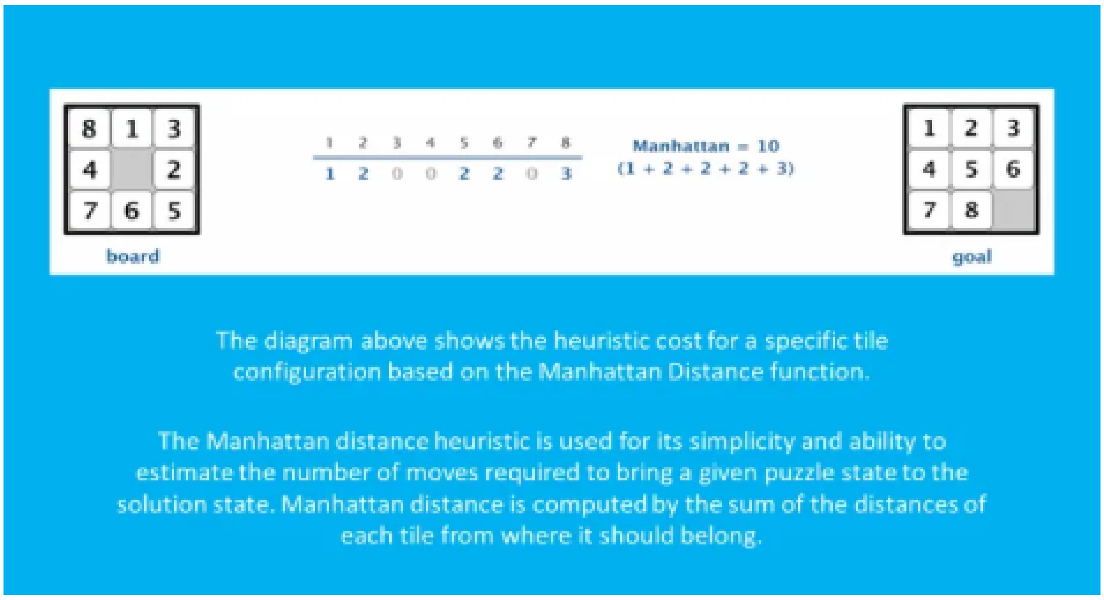
A* search is a computer search algorithm widely used for pathfinding and graph traversal. In our case of the 8-puzzle problem, we will use it for optimal graph traversal.

THE COST FUNCTION OF AN 8-PUZZLE STATE

The cost value of an 8-puzzle state is a combination of two values. It is often called the cost function f. where h the heuristic cost gives how far the goal node is, and g is the number of nodes traversed from the start node to the current node. For h, we will use the Manhattan distance, and for g, we will use the depth of the current node.

MANHATTAN DISTANCE

For our A* search, I will use the sum of the Manhattan distance and the current depth of the node as the total cost. Manhattan distance heuristic for its simplicity and ability to estimate the number of moves required to bring a given puzzle state to the solution state. We compute this distance by the sum of the lengths of each tile from where it should belong.



The diagram above shows the heuristic cost for a specific tile configuration based on the Manhattan Distance function.

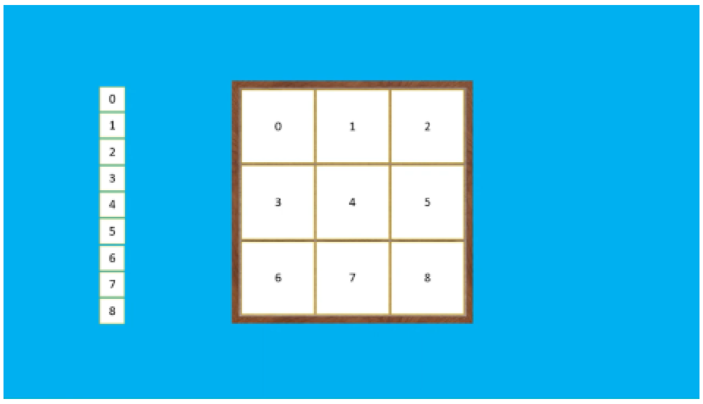
The Manhattan distance heuristic is used for its simplicity and ability to estimate the number of moves required to bring a given puzzle state to the solution state. Manhattan distance is computed by the sum of the distances of each tile from where it should belong.

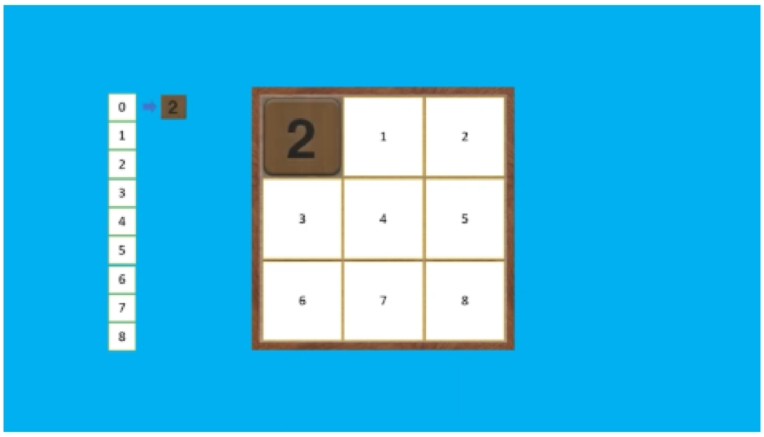
THE PUZZLE STATE

To solve the 8-Puzzle problem, we need a data structure to represent the puzzle's tiles, which we will refer to as the puzzle's state. Each state represents a unique combination of tiles, and throughout our solving process, we will need to manage potentially hundreds or thousands of these states. Each distinct tile arrangement in the puzzle corresponds to a node within a tree data structure.

This picture shows the neighbouring indices for index 6 of the integer array. On the right it shows the neighbouring index array for all the indices of the integer array. Summarise this relationship in a list of neighbour indices for each tile index.

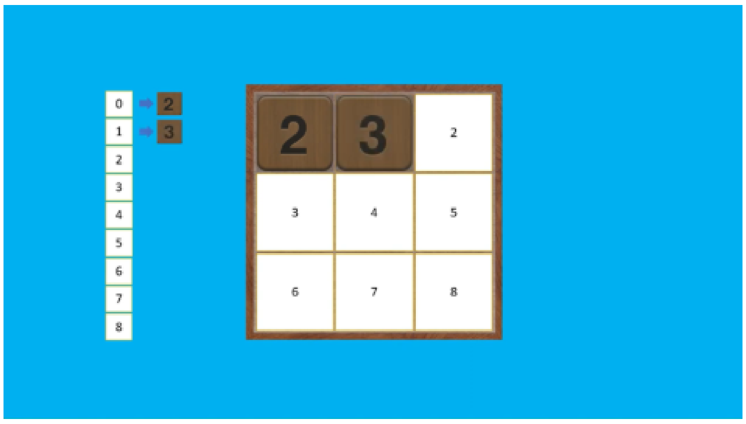
- **Index 0: {1, 3} , Index 1: {0, 2, 4}, Index 2: {1, 5}, Index 3: {4, 0, 6}**
- **Index 4: {3, 5, 1, 7}, Index 5: {4, 2, 8}, Index 6: {7, 3}, Index 7: {6, 8, 4}**
- **Index 8: {7, 5}**

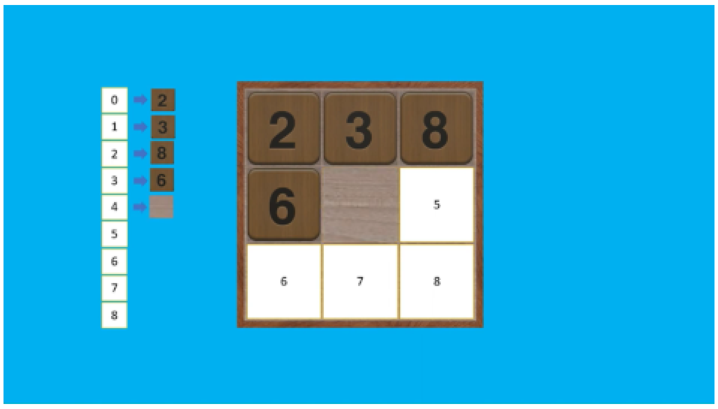


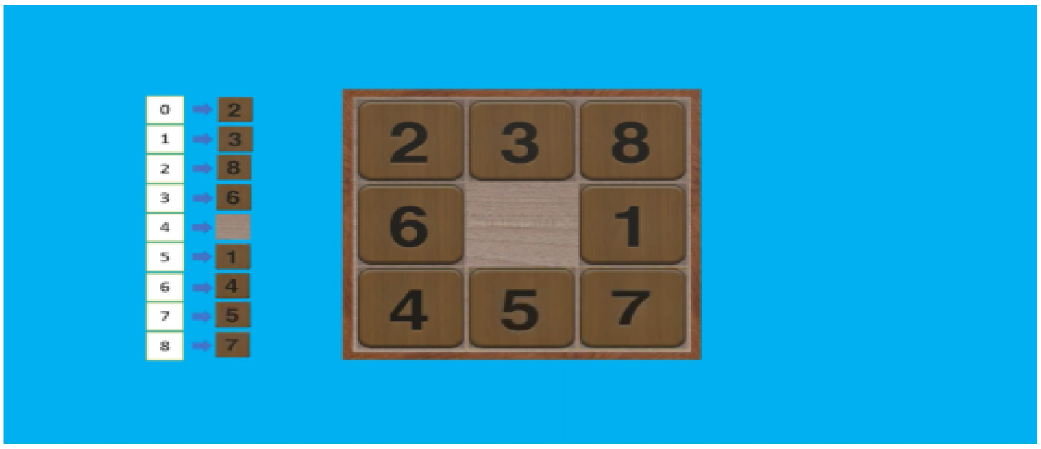


In this picture, you can see that the relationship
between the index of the integer array and the placement of tiles in the 8-Puzzle.

To represent these states, we will use an integer array whose indices correspond to specific tile positions.
The values stored at these indices represent the tile numbers. In this one-dimensional array representation, each index is fixed and represents a predefined tile location. This precise representation is crucial for our problem-solving process.







In this picture, you can see that the fourth index, which is 4, in the integer array is now set to the empty tile of the 8-Puzzle.

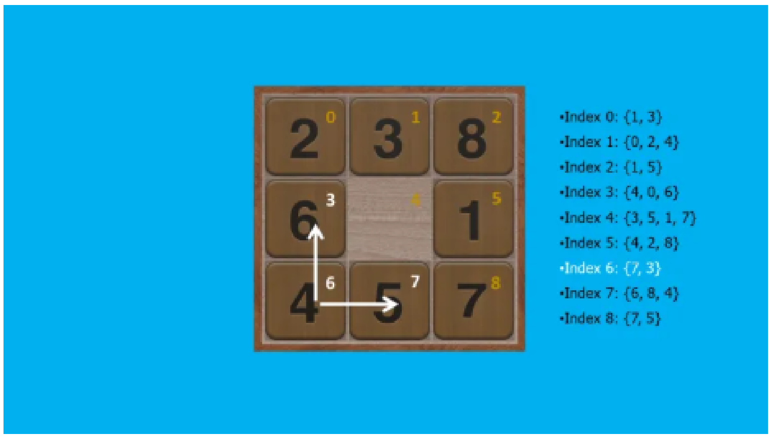
By manipulating the values in this array representation, while adhering to the constraint of where the empty tile can move with each action, we can efficiently progress towards reaching the goal state of the puzzle.
This approach allows us to efficiently track and navigate through the various configurations of the puzzle during the solving process, making our solution approach highly effective.

THE NEIGHBOURS

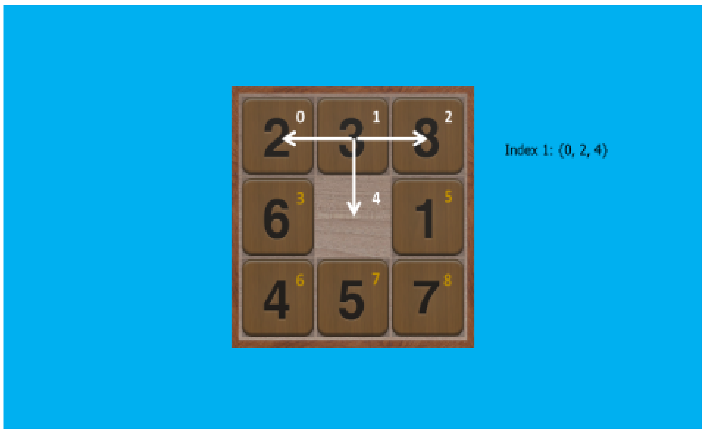
Our next objective is to construct the graph of neighbours for the 8-Puzzle problem. This involves determining the neighbouring indices, for each of the 9 indices.

Constructing the graph of neighbours involves determining the possible moves or adjacent positions (neighboring indices) that the empty space (representing the movable tile) can move to from each tile position (or index) within the puzzle grid. In the context of the 8-Puzzle:

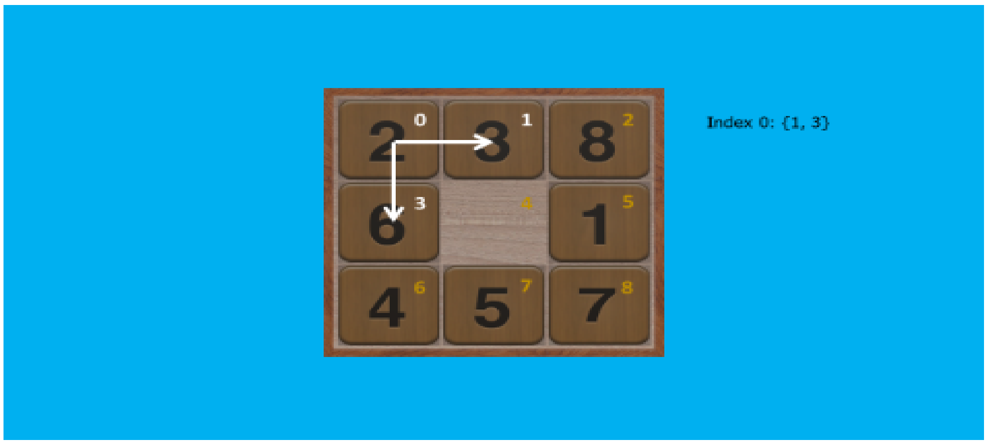
1. We have a 3×3 grid where each cell is represented by an index ranging from 0 to 8, as described in the PuzzleState.
2. Now, the task is to identify, for each index (representing a tile position), which other indices (tile positions) are its neighboring positions, meaning where the empty space can move to from that particular position.
3. By constructing this graph of neighbours, we establish the valid moves or transitions between different states of the puzzle, enabling the A* search to efficiently navigate the puzzle space and find the optimal sequence of moves to reach the goal configuration from any given start configuration.



- Index 0: {1, 3}
- Index 1: {0, 2, 4}
- Index 2: {1, 5}
- Index 3: {4, 0, 6}
- Index 4: {3, 5, 1, 7}
- Index 5: {4, 2, 8}
- Index 6: {7, 3}
- Index 7: {6, 8, 4}
- Index 8: {7, 5}



Index 1: {0, 2, 4}



Index 0: {1, 3}

This graph representation is fundamental for implementing search algorithms in solving the 8-Puzzle problem.