*Name: Ketan Madan Joshi*

*Student ID: - 1017922060*

*Project 1*

**Iterative Deepening Search (IDS) Implementation:**

* The code successfully implements the IDS algorithm for solving the 8-puzzle problem.
* It uses a depth-first search strategy with an iterative deepening approach, gradually increasing the depth limit until a solution is found.
* Neighbor Generation Function (neighbours):

The neighbours function generates valid neighboring states for a given puzzle state, contributing to the exploration of the state space during search.

* Depth-First Search (depth\_first\_search):

The depth\_first\_search function performs recursive depth-first search, exploring the puzzle state space and backtracking when necessary.

* It prints the final path of empty tile positions when a solution is found.
* IDS Termination and Result Handling:

The code appropriately terminates the IDS search when a solution is found within the specified depth limit.

* It returns the final path of empty tile positions, providing insight into the sequence of moves leading to the solution.

**Limitations of the Implemented Code:**

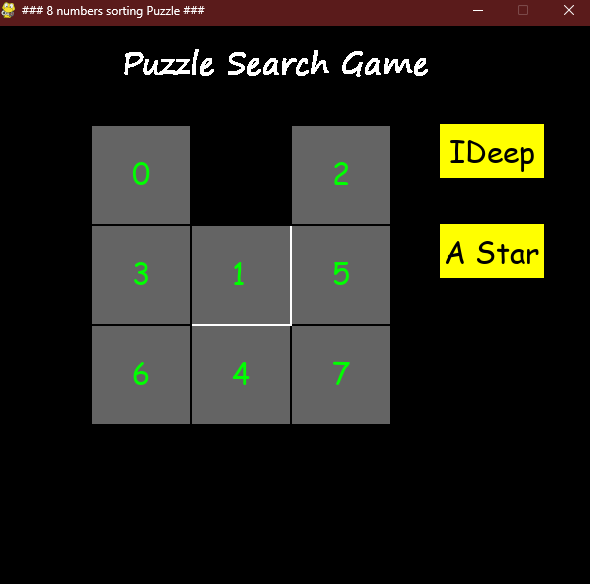
* Fixed Depth Limit:

The IDS implementation uses a fixed maximum depth (max\_depth = 20). This limitation may impact the ability to find solutions for certain instances of the puzzle, especially those requiring deeper exploration.

* Limited Input Handling:

The code lacks user interface elements and input handling mechanisms. It currently operates as a programmatic solution, restricting its interactive use by end-users.

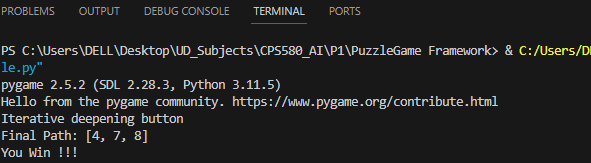
**Initial State:**



**Goal state:**



**Final path for IDS:**



**A\* Algorithm Implementation:**

* The code successfully implements the A\* algorithm, for solving the 8-puzzle problem.
* It combines both uniform cost search and a heuristic function to efficiently explore the state space.
* Heuristic Function (heuristic):

The code includes a heuristic function that estimates the cost from the current state to the goal state.

The heuristic function is implemented to count the number of misplaced tiles, providing a simple yet functional estimate.

* Priority Queue Handling:

A priority queue efficiently manages nodes based on their priority values, incorporating both the cost to reach the state and the heuristic estimate.

The use of a priority queue allows the algorithm to prioritize exploring states with lower estimated costs, contributing to optimality.

* Path Printing and Return:

When the goal state is reached, the code prints the final path of empty tile positions and returns it as a result.

* The path provides insight into the sequence of moves leading to the solution.

**Limitations of the Implemented A\* Search:**

* Heuristic Dependency:

The effectiveness of A\* heavily depends on the quality of the heuristic function. The provided heuristic is simplistic and may not always accurately represent the actual cost to reach the goal.

* Memory Usage:

A\* can demand significant memory resources, especially in scenarios with large state spaces. Storing and managing the priority queue can become a challenge, potentially impacting performance on memory-constrained systems.

* Informed Search Constraints:

While A\* is an informed search algorithm, it relies on problem-specific information. In scenarios where such information is limited or unavailable, A\* might not perform optimally.

* No Dynamic Depth Adjustment:

Similar to the IDS implementation, the depth of exploration is not dynamically adjusted based on the characteristics of the puzzle instance.

**Initial State:**

A screenshot of a game

Description automatically generated

**Goal state:**

A screenshot of a game

Description automatically generated

**Final path for A\*:**

A screen shot of a computer

Description automatically generated