Tile Solver Game

Team Members

- Ousman Bah
- SHAWN
- Eduardo A. Paramio

Project Overview

Welcome to the **Tile Solver Game**! This is a Streamlit-based web application that allows users to upload images, convert them into playable 8-puzzle games, and solve them using the A* algorithm.

Installation & Setup

Prerequisites

- Python 3.8 or higher
- pip package manager

Installation Steps

1. Clone or download this repository
2. Navigate to the project directory
3. Install required dependencies:

pip install -r requirements.txt

Running the Application

streamlit run app.py

Features

Core Functionality

1. **Image Upload**: Upload JPG, PNG, or JPEG images
2. **Puzzle Generation**: Automatically splits uploaded images into 3x3 tile puzzles
3. **Manual Play**: Interactive controls to manually solve the puzzle
4. **Shuffle Function**: Randomize puzzle tiles with adjustable difficulty
5. *A Algorithm Solver**: Automatic optimal solution finder

User Interface Sections

- **Shuffle Section**: Display puzzle + shuffle controls
- *A Solver Section**: Display puzzle + automatic solving capability
- **Manual Controls**: Directional buttons for manual gameplay

🤖 A* Algorithm Implementation

Algorithm Details

- **Search Method**: A* pathfinding algorithm
- **Heuristic Function**: Manhattan Distance
- **Optimality**: Guaranteed to find the shortest solution path
- **Time Complexity**: O(b^d) where b is branching factor, d is solution depth
- **Space Complexity**: O(b^d)

Heuristic Explanation

The Manhattan Distance heuristic calculates the minimum number of moves required for each tile to reach its goal position. This is computed as the sum of horizontal and vertical distances between current and target positions for all tiles.

Why Manhattan Distance?

- Admissible: Never overestimates the actual cost
- Consistent: Satisfies the triangle inequality
- Efficient: Fast to compute
- Optimal for grid-based sliding puzzles

Algorithm Flow

1. Convert puzzle state to hashable tuple format
2. Initialize priority queue with start state
3. Use f(n) = g(n) + h(n) where:
    - g(n) = actual cost from start
    - h(n) = Manhattan distance heuristic
4. Explore neighbors by moving blank tile
5. Reconstruct optimal path when goal reached

🔧 Technical Implementation

Core Components

Image Processing (`utils.py`)

- `split_image()`: Divides uploaded image into 3x3 grid tiles
- `pil_to_base64()`: Converts PIL images to base64 for web display
- `shuffle_board()`: Randomizes puzzle state with valid moves
- `is_solved()`: Checks if puzzle is in solved state

A* Algorithm (`a_star.py`)

- `a_star()`: Main search algorithm implementation
- `manhattan_distance()`: Heuristic function calculation
- `get_neighbors()`: Valid move generation
- `solve_with_astar()`: User interface integration

User Interface (`manual_play.py`, `app.py`)

- Streamlit-based web interface
- Real-time puzzle state display
- Interactive controls for manual gameplay
- Results visualization for A* solutions

Data Structures

- **Puzzle State**: 3x3 matrix with None representing blank space
- **Priority Queue**: heapq for A* open set management
- **State Tracking**: Dictionaries for came_from and g_score values

🎮 How to Use

1. **Upload Image**: Choose any JPG, PNG, or JPEG file
2. **Shuffle Puzzle**: Use the slider to set difficulty, click "Shuffle Now"
3. **Manual Play**: Use directional buttons (Up, Down, Left, Right) to move tiles
4. **Auto Solve**: Click "Solve Current Puzzle with A*" to see optimal solution

📊 A* Algorithm Output

When A* finds a solution, it displays:

- Total number of moves required
- Step-by-step move sequence in (row, col) → (row, col) format
- Coordinates use zero-based indexing