# Giotto.AI Assessment Task

Salvatore Pascarella

April 2025

## 1 Code Structure

### 1.1 `config.py`

It contains key constants used for visualization, component filtering, and symbolic printing.

- COLOR: colors used to plot the final grid.
- SYMBOLS: a dict used to finally visualize the grid through the terminal.
- MIN_COMPONENT_SIZE = 8: Minimum number of pixels required for a component to be evaluated as a candidate for a closed shape. This value filters out noise or single pixels. A size of 8 corresponds to the smallest possible component which would be a 3x3 square

### 1.2 `plot.py`

It provides functions to print, visualize, and verify the predicted grid output againts the ground truth.

### 1.3 `solve.py`

Before starting the implementation, I analyzed the training grids to identify shared structural patterns. A key observation was that non-closed shapes remain unchanged in the output, whereas closed shapes are transformed. The transformation consisted in filling the first inner layer of closed-shaped components with green pixels while the first outer layer with red pixels (ring-like). Therefore, the first step is to identify the connected components of 1s. To achieve this, I implemented a DFS that explores all 4-connected neighbors (up, down, left, right) from each unvisited cell with value 1, grouping all reachable cells into a single component. **Why DFS?** I chose DFS for its simplicity, efficiency, and suitability for grid-based traversal. Since the grids are small, recursion depth is not an issue. DFS also requires minimal overhead, no auxiliary data structures are needed beyond a visited boolean variable and it naturally captures the full extent of each connected region. The function `find_components` implements this and returns a list of components, where each component is a list of (i, j) coordinate pairs. After identifying components, I used:

- `get_bounding_box(component)` to compute the minimum and maximum row and column indices that define the rectangular boundary of a shape.
- `extract_subgrid(grid, bbox)` which extracts the subgrid that spans the bounding box.

To determine if a component is a closed shape, I implemented a flood fill algorithm on the subgrid. `flood_fill_zeros(subgrid)` starts from all 0s on the border of the subgrid and marks all reachable background cells using BFS. If there exist 0s inside the subgrid that are not reachable from the border, they are considered enclosed. This is the core for the `is_closed_shape()` function. For each component classified as closed, `fill_inner_area()` marks all inner 0s adjacent to the shape's interior with 3s (green pixels) while `fill_outer_area()` marks all background pixels immediately surrounding the shape's outer edge with 2s (red pixels). To be honest, I used GPT to help me translate into code the flood fill algorithm. The transformed output is saved via `save_grid()`, which stores the processed grid as a formatted JSON file. Finally, `check_and_plot()` compares the result with the ground truth and visualizes the grid. Figure 1 shows the predicted output grid on the test grid.

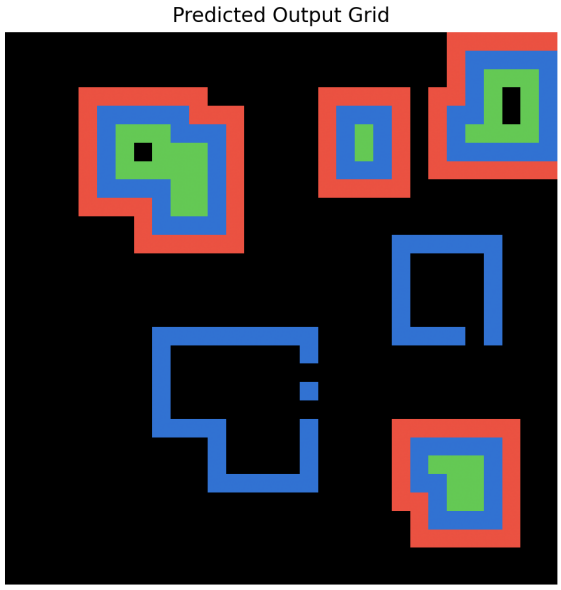


Figure 1: Predicted Output Grid