CS 4100 5100

# Shapeshifting Coloring Problems[*]

## Instructions

**ALERT:** This assignment may carry a high workload, depending on your programming experience. You will need to make several changes to your code for your approach to meet expectations, and succeed on the autograder. **Start early, and read all instructions!**

## Academic Integrity

- If you discuss concepts related to this programming assignment with one or more classmates, all parties must declare collaborators in their individual submissions within a code comment. Such discussions must be kept at a conceptual level, and no sharing of actual code is permitted.
- You may use any generative AI tool available to you, as long as it is appropriately cited in a code comment. I recommend using AI tools for debugging or conceptual understanding rather than to produce actual functions.
- Failure to disclose collaborations or use of generative AI will be treated as a violation of academic integrity, and penalties listed in the course syllabus will be enforced.

## Reach Out!

If at any point you feel stuck with the assignment, please reach out to the TAs or the instructor, and do so early on! This lets us guide you in the right direction in a timely fashion and will help you make the most of your assignment.

[*]Developed by Ashwin Bharadwaj, Anio Zhang and Rajagopal Venkatesaramani.
Model AI Assignments, EAAI 2025.

In this assignment, you are given a PyGame environment (in the `gridgame.py` file) that renders a randomly initialized $n \times n$ grid, with some cells pre-filled with one of four colors. Your goal is to build an AI agent that uses **local search** to solve a coloring problem over this grid (see pages 3 and 4 for constraints), such that no two cells that share an edge have the same color, while also minimizing the number of shapes and colors used.

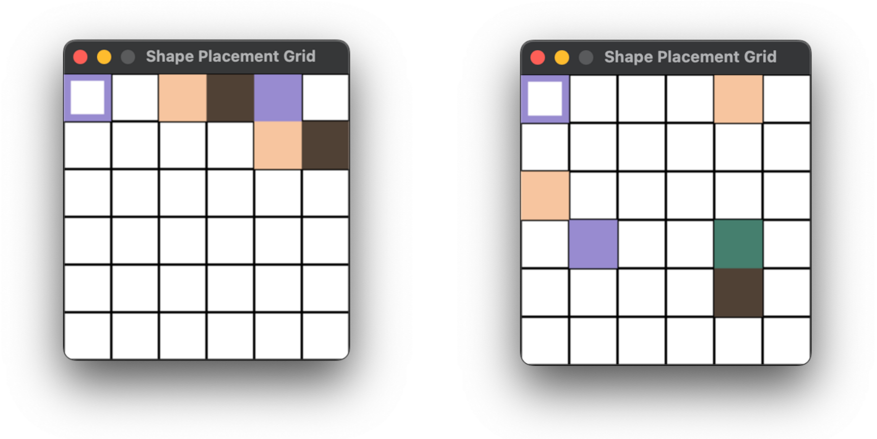
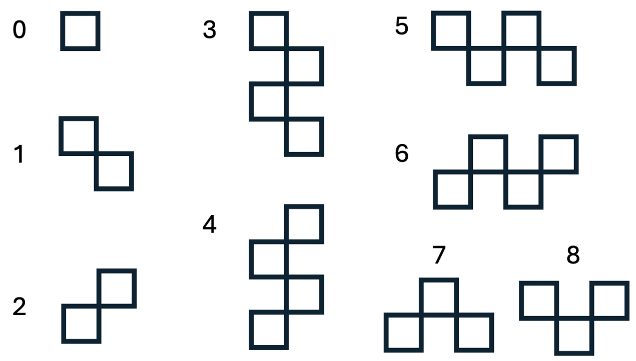


Figure 1: Examples of Initial Configurations

Your agent will attempt to fill the environment by moving a virtual 'brush' over this grid and placing colored shapes, where the shape of the brush can be cycled through the following choices (numbered 0-8):



Additionally, each brush can be cycled through one of four colors[*] (numbered 0-3).

🟪 0. Indigo

🟫 1. Taupe

🟩 2. Viridian

🟧 3. Peach

[*]If, despite our best efforts, the color choices here pose accessibility concerns, they may be edited in the `gridgame.py` file. Alternatively, please reach out to us, and we'd be happy to help edit them!

Your agent may interact with the grid environment **only** using the `execute()` function called from within the `hw1.py` file, with the following argument options (passed as strings):

- `export`: returns the current state of the grid, a list of shapes with positions and colors currently placed on the grid, and a Boolean indicating whether the coloring constraints have been satisfied.
- `up/down/left/right`: move the brush in the specified direction by one cell. The brush starts in the top left corner of the grid when the program is executed.
- `place`: place a shape on the grid, i.e. color the cells covered by the brush in the currently selected brush color.
- `switchshape`: cycle to the next brush shape option.
- `switchcolor`: cycle to the next brush color option.
- `undo`: undo the last placed shape.

Running the `execute()` function with any argument returns six items:

| **Variable** | **Data** | **Description** |
|---|---|---|
| `shapePos` | **Brush Position** | Current location of the brush, (X, Y coordinates) `list` of size 2 |
| `currentShapeIndex` | **Current Shape Index** | Index of the currently selected shape, `int` |
| `currentColorIndex` | **Current Color Index** | Index of the currently selected color, `int` |
| `grid` | **Grid State** | Updated state of the ($n \times n$ grid), `np.array` (shape $n \times n$) |
| `placedShapes` | **Placed Shapes** | List of shapes already placed on the grid, `list(int)` |
| `done` | **Coloring Constraints Satisfied?** | Boolean indicating if coloring constraints are met, `bool` |

Table 1: Summary of Data in the Shapeshifting Coloring Environment

You may use any public helper functions defined within `gridgame.py` by calling them in the `hw1.py` file, but you may **not** use any of the private functions (private functions contain an underscore, i.e., '_' at the start of the function name). Additionally, you may access (look up) grid attributes such as `game.grid, game.gridSize, game.colors` etc., but you may **not** modify them directly by setting/assigning a desired value. Any modification to the grid environment must only be achieved through the `execute()` command.

Your goal is to build an AI agent that colors this grid, such that no two adjacent cells share the same color. Further, your goal is to achieve this coloring using as few shapes and colors as possible - a larger brush may cover multiple cells, but counts as one shape. While implementing a baseline agent using basic search techniques such as BFS or a greedy approach would be a good way to get used to the environment, your final submission **must use a local search** approach such as hill climbing, simulated annealing or genetic algorithms to implement your agent. Since a local search may not always converge to the 'optimal' coloring, this assignment is graded on correctness, rather than optimality.

**To receive full points, your implementation must** a) use a local search approach, b) complete the board with no constraint violations, c) reasonably attempt to minimize the number of colors and shapes used, d) follow all the code rules specified below, and e) on average, lead to a valid coloring of 2 test cases within 10 minutes.

**NOTE:** The autograder only checks the validity of the final solution. The autograder score is not your final grade - points will manually be deducted if any of the rules/expectations outlined above were not followed in your submission.

**Environment Rules:** Adjacent cells are defined as cells that share an edge between them (i.e., diagonally neighboring cells may share the same color, since they only share a vertex). If a brush partially or fully overlaps with an area of the grid that is already colored, the `execute` function with the `place` argument will fail, i.e. the cells will not be overwritten.

The environment is built as a Python class. You are only allowed to interact with the environment with the `.execute()` method. Do not use any other method from the class to modify any variables in the class. However, you can use the provided public functions (i.e., the `canPlace()`, `getAvailableColors()` and `checkGrid()` functions) to help you write your solution. Feel free to change the arguments for the constructor in `hw1.py` to make the exercise easier for debugging or harder for a personal challenge.

**Assignment Rules:** Any hardcoded solutions, or attempts to leverage the autograder's design to maximize points scored will result in an automatic zero on the assignment. Your agent **must** only use the `execute()` function to interact with the environment using its different arguments, and use its returned values to implement your objective function, or any validity check you may wish to use. Do not directly manipulate the `gridgame` variables. Treating the environment as a black box (even when you know its internals) is a very important concept in AI and will serve you well in future assignments, as well as in your careers. Implementations must be optimized for good runtime - Gradescope has an autograder timeout of 10 minutes. Any submissions that do not finish executing in 10 minutes will be treated as incomplete, and will be evaluated for partial credit based on correctness.

**Submission:** Upload your completed `hw1.py` file, and the provided `gridgame.py` file to Gradescope, and ensure the autograder testcases run as intended with no errors. Please make sure you do not rename the files, or zip a directory containing the files; upload the two files directly to Gradescope instead.

## Beyond the Assignment - Broader Applications

Graph coloring is a versatile problem in graph theory with numerous practical applications across various fields. Here are some example problems from the real world that are mathematically equivalent to graph coloring.

### 1. Scheduling

- In exam scheduling, vertices may represent exams and edges connect exams with common students. Colors represent time slots. A proper coloring ensures no student has conflicting exams. Ideas similar to the different brush types in this assignment may be used to optimize room usage, time between exams, etc.
- Graph coloring can be used to schedule classes, ensuring no conflicts in room assignments or student schedules. Faculty room assignments for teaching courses often follow a similar process.
- Graph coloring can optimize aircraft maintenance schedules, minimizing downtime and maximizing efficiency.

### 2. Resource Allocation

- In radio and mobile networks, graph coloring can be used to assign specific frequencies to a set of transmitters, while the objective function tries to minimize interference. Vertices represent transmitters, edges may represent potential interference as a result of a given frequency allocation, and colors represent frequencies.
- Compilers use graph coloring to allocate registers efficiently. Variables form vertices, edges connect variables that are simultaneously active. Colors represent registers.
- Sudoku puzzles are a resource allocation problem that can be modeled as graph coloring problems. Each cell is a vertex, with edges connecting cells that cannot have the same number. Colors represent the numbers 1-9.

### 3. Network Analysis

- In various network scenarios, graph coloring can be used to identify and resolve conflicts, such as in wireless sensor networks or traffic management systems.
- Graph coloring may be used in cybersecurity and network security applications.