

|  |
| --- |
| **RAJALAKSHMI INSTITUTE OF TECHNOLOGY** |
| (An Autonomous Institution, Affiliated to Anna University, Chennai) |

**DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE**

**ACADEMIC YEAR 2025 - 2026**

**SEMESTER III**

**ARTIFICIAL INTELLIGENCE LABORATORY**

**MINI PROJECT REPORT**

|  |  |
| --- | --- |
| **REGISTER NUMBER** | 2117240070125 |
| **NAME** | Jaya Sonica |
| **PROJECT TITLE** | AI-Based Map Coloring Game Using Flood Fill Algorithm |
| **DATE OF SUBMISSION** | 4.11.25 |
| **FACULTY IN-CHARGE** | **Mrs. M. Rubina begam** |

**Signature of Faculty In-charge**

**INTRODUCTION**

Artificial Intelligence (AI) focuses on developing systems that can simulate human reasoning and problem-solving. One such problem is **map coloring**, where regions on a map must be filled with colors so that no two adjacent areas share the same color.

This project presents a **Coloring Game** based on the **map coloring concept** and implemented using the **flood fill algorithm**. The system allows users to interactively color regions on a map while automatically checking adjacency constraints. It demonstrates how AI concepts and algorithms can be applied in an engaging and visual way to solve constraint-based problems.

**PROBLEM STATEMENT**

In traditional map coloring problems, the goal is to color each region of a map such that no two adjacent regions share the same color, which can be time-consuming and error-prone when done manually. This project aims to develop an **AI-based interactive coloring game** that automatically enforces the map coloring rule using the **flood fill algorithm**. The system allows users to select and fill colors in different regions while preventing the use of the same color for neighboring areas, effectively demonstrating the **constraint satisfaction** concept of Artificial Intelligence in an engaging, visual, and user-friendly manner.

**GOAL**

The goal of this project is to develop an AI-based interactive map coloring system that ensures no two adjacent regions share the same color. The system aims to provide a user-friendly interface where users can select and apply colors to different regions of a map while the AI automatically checks and enforces the map coloring rule using the flood fill algorithm. The expected result is a visually appealing, constraint-satisfied colored map that demonstrates the principles of artificial intelligence and constraint satisfaction. Additionally, the project opens possibilities for educational use, game-based learning, and real-world applications such as territory mapping, graph coloring, and resource allocation visualization.

**THEORETICAL BACKGROUND**

The map coloring problem is a classic example of a **Constraint Satisfaction Problem (CSP)** in Artificial Intelligence, where the goal is to assign colors to regions of a map such that no two adjacent regions share the same color. It represents real-world applications like scheduling, resource allocation, and geographical zoning. The **flood fill algorithm**, used in this project, is a fundamental computer graphics algorithm that determines and fills connected areas with a chosen color. It works by identifying all pixels connected to a starting point that share the same color and replacing them with a new one.

In the **literature**, several algorithms have been explored for map coloring, including **backtracking**, **graph coloring using greedy algorithms**, and **genetic algorithms** for optimization. While these methods are efficient for solving large-scale CSPs or optimizing color usage, they often involve complex logic and higher computational overhead.

The **flood fill algorithm** is chosen for this project due to its simplicity, efficiency, and suitability for interactive graphical applications. It allows real-time color filling and visual feedback, making it ideal for a user-driven coloring game. Additionally, it provides an intuitive way to demonstrate constraint satisfaction, ensuring an engaging and educational experience while maintaining computational efficiency.

**ALGORITHM EXPLANATION WITH EXAMPLE**

**Algorithm:**

1. Start from a seed point (x, y) inside the region.
2. Determine the target color (the color to replace) and the replacement color (new color).
3. If the current pixel color matches the target color, change it to the new color.
4. Add the neighboring pixels (up, down, left, right) to a list or queue for further checking.
5. Continue the process until all connected pixels with the target color are replaced.

**Example:**  
Consider a simple 2D grid representing a map:

R R G

R G G

B B G

Where **R = Red**, **G = Green**, **B = Blue**.  
If the flood fill is applied starting at position (0,0) (top-left corner) with **Y (Yellow)** as the new color, the algorithm checks all connected red cells and replaces them with yellow. The updated grid becomes:

Y Y G

Y G G

B B G

This demonstrates how flood fill identifies and fills connected regions efficiently.

**Justification for Algorithm Selection:**  
The flood fill algorithm is chosen because it provides a **simple yet effective** way to implement region-based coloring in an interactive system. It ensures **accurate boundary detection**, supports **real-time user interaction**, and visually demonstrates how AI algorithms can handle **constraint satisfaction** dynamically. It is both intuitive for visualization and efficient for small to medium-sized maps used in educational or game-based applications.

**IMPLEMENTATION AND CODE**

import pygame, sys

from collections import deque

pygame.init()

WIDTH, HEIGHT = 900, 700

WIN = pygame.display.set\_mode((WIDTH, HEIGHT))

pygame.display.set\_caption("🎨 Smart Map Coloring")

# Load and scale map

MAP\_IMAGE = pygame.image.load("sketch.png").convert()

MAP\_IMAGE = pygame.transform.scale(MAP\_IMAGE, (WIDTH, HEIGHT))

DRAW\_SURFACE = MAP\_IMAGE.copy()

# Colors palette

COLORS = [

    (255, 0, 0), (0, 255, 0), (0, 0, 255),

    (255, 255, 0), (255, 165, 0), (128, 0, 128),

    (0, 255, 255), (255, 192, 203), (139, 69, 19),

    (210, 180, 140), (0, 0, 0), (255, 255, 255)

]

FONT = pygame.font.SysFont("Comic Sans MS", 26, bold=True)

selected\_color\_index = 0

warning\_message = ""

regions = []

next\_region\_id = 1

def color\_close(c1, c2, tolerance=30):

    return all(abs(c1[i] - c2[i]) < tolerance for i in range(3))

def get\_region\_pixels(surface, x, y, tol=30):

    target = surface.get\_at((x, y))[:3]

    q = deque([(x, y)])

    visited = set()

    while q:

        cx, cy = q.popleft()

        if (cx, cy) in visited:

            continue

        if cx < 0 or cy < 0 or cx >= WIDTH or cy >= HEIGHT:

            continue

        cur = surface.get\_at((cx, cy))[:3]

        if not color\_close(cur, target, tol):

            continue

        visited.add((cx, cy))

        q.extend([(cx+1, cy), (cx-1, cy), (cx, cy+1), (cx, cy-1)])

    return visited

def fill\_pixels(surface, pixels, fill\_color):

    for (px, py) in pixels:

        surface.set\_at((px, py), fill\_color)

def get\_centroid(pixels):

    if not pixels:

        return (0, 0)

    sx = sum(p[0] for p in pixels)

    sy = sum(p[1] for p in pixels)

    return (sx // len(pixels), sy // len(pixels))

def regions\_touch(pixels\_a, pixels\_b):

    bset = set(pixels\_b)

    for (x, y) in pixels\_a:

        if ((x+1, y) in bset) or ((x-1, y) in bset) or ((x, y+1) in bset) or ((x, y-1) in bset):

            return True

    return False

def palette\_rects():

    rects = []

    cols\_per\_row = 8

    swatch\_size = 50

    padding = 10

    start\_x = 50

    start\_y = HEIGHT - 150

    for i in range(len(COLORS)):

        row = i // cols\_per\_row

        col = i % cols\_per\_row

        x = start\_x + col \* (swatch\_size + padding)

        y = start\_y + row \* (swatch\_size + padding)

        rects.append((i, pygame.Rect(x, y, swatch\_size, swatch\_size)))

    return rects

def draw\_palette():

    for i, rect in palette\_rects():

        pygame.draw.rect(WIN, COLORS[i], rect)

        if i == selected\_color\_index:

            pygame.draw.rect(WIN, (0, 0, 0), rect, 3)

def reset\_map():

    global DRAW\_SURFACE, warning\_message, regions, next\_region\_id

    DRAW\_SURFACE = MAP\_IMAGE.copy()

    warning\_message = ""

    regions = []

    next\_region\_id = 1

def draw\_screen():

    WIN.blit(DRAW\_SURFACE, (0, 0))

    title = FONT.render("🎨 Click to color | Click palette to choose | R to reset | ESC to exit", True, (0, 0, 0))

    WIN.blit(title, (40, 20))

    draw\_palette()

    if warning\_message:

        warn = FONT.render(warning\_message, True, (255, 0, 0))

        WIN.blit(warn, (40, 60))

    pygame.display.update()

# ---------------- Main Loop ----------------

clock = pygame.time.Clock()

while True:

    clock.tick(30)

    draw\_screen()

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            pygame.quit(); sys.exit()

        elif event.type == pygame.KEYDOWN:

            if event.key == pygame.K\_ESCAPE:

                pygame.quit(); sys.exit()

            elif event.key == pygame.K\_r:

                reset\_map()

        elif event.type == pygame.MOUSEBUTTONDOWN:

            pos = event.pos

            clicked\_palette = None

            for i, rect in palette\_rects():

                if rect.collidepoint(pos):

                    clicked\_palette = i

                    break

            if clicked\_palette is not None:

                selected\_color\_index = clicked\_palette

                warning\_message = ""

                continue

            if pos[0] < 0 or pos[1] < 0 or pos[0] >= WIDTH or pos[1] >= HEIGHT:

                continue

            new\_region\_pixels = get\_region\_pixels(MAP\_IMAGE, pos[0], pos[1])  # region from original map

            if not new\_region\_pixels or len(new\_region\_pixels) < 5:

                warning\_message = "Click inside a valid region!"

                continue

            new\_color = COLORS[selected\_color\_index]

            violates = False

            # Check adjacency first before coloring

            for reg in regions:

                if reg['color'] == new\_color:

                    if regions\_touch(new\_region\_pixels, reg['pixels']):

                        violates = True

                        break

            if violates:

                warning\_message = "❌ Adjacent region already has this color!"

            else:

                fill\_pixels(DRAW\_SURFACE, new\_region\_pixels, new\_color)

                centroid = get\_centroid(new\_region\_pixels)

                regions.append({

                    'id': next\_region\_id,

                    'color': new\_color,

                    'pixels': new\_region\_pixels,

                    'centroid': centroid

                })

                next\_region\_id += 1

                warning\_message = ""

**OUTPUT**

****

**Figure 1: The user colors the snowman’s hat with a cyan shade.**

****

**Figure 2: The system prevents applying the same color to an adjacent region and displays a warning.**

****

* **Figure 3: The user selects a new color (pink) and successfully colors the next region without conflict.**

**RESULTS AND FUTURE ENHANCEMENT**

The developed **AI-based Map Coloring Game** effectively demonstrates how the **Flood Fill algorithm** can be applied to solve the **map coloring problem** while maintaining the rule that no two adjacent regions share the same color. The system provides an intuitive and interactive interface, ensuring accurate color filling and immediate feedback when a constraint is violated.

Compared to traditional manual coloring methods or simple paint tools, this approach integrates **artificial intelligence logic** to automate rule enforcement, making it both educational and engaging.

In the future, the project can be enhanced by:

* Implementing **automatic color assignment** using graph coloring algorithms.
* Adding **machine learning** to identify and separate regions automatically.
* Providing **score-based gameplay** or challenges for educational purposes.

These improvements would make the system more versatile, intelligent, and suitable for both learning and entertainment applications.

|  |  |
| --- | --- |
| **Git Hub Link of the project and report** | [**ai-mini-project/ at main · jayasonicas/ai-mini-project**](https://github.com/jayasonicas/ai-mini-project/tree/main) |

**REFERENCES**

* Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 4th Edition, Pearson Education, 2021.
* Mark Lutz, *Programming Python*, 5th Edition, O’Reilly Media, 2020.
* *Pygame Official Documentation*, Available at: https://www.pygame.org/docs/
* GeeksforGeeks, “Flood Fill Algorithm – Explained with Examples”, Available at: https://www.geeksforgeeks.org/flood-fill-algorithm/
* TutorialsPoint, “Python Graphics Programming with Pygame”, Available at: https://www.tutorialspoint.com/python\_pygame/index.htm