

|  |
| --- |
| **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** | 2117240070027 |
| **NAME** | ARUNAGIRI.R |
| **PROJECT TITLE** | Graph Coloring Puzzle Game using Map Coloring Algorithm (GUI Implementation) |
| **DATE OF SUBMISSION** |  |
| **FACULTY IN-CHARGE** | **Mrs. M. Divya** |

**Signature of Faculty In-charge**

**INTRODUCTION**

Artificial Intelligence enables machines to mimic human reasoning and problem-solving. One of its core applications is constraint satisfaction, where a system must assign values (or colors) under specific limitations.  
The **Map Coloring Problem** is a classic example of a constraint satisfaction problem (CSP), where no two adjacent regions on a map should share the same color. This concept is widely used in AI, optimization, and even scheduling problems.  
This project presents a **Graph Coloring Puzzle Game** with an interactive **Graphical User Interface (GUI)** built using Python’s **Tkinter** library. Users can choose shapes (Triangle, Square, House, Star, Hexagon, etc.) and color each node ensuring adjacent nodes do not share the same color. It visually demonstrates backtracking and constraint-checking logicBrief overview of Artificial Intelligence concepts.

Clearly introduce the topic, provide background context, explain why the problem matters, and highlight what the project aims to accomplish.

**PROBLEM STATEMENT**

* To develop a **GUI-based puzzle game** that allows users to color nodes of a graph such that no two connected nodes have the same color. The system should dynamically check for conflicts and provide hint or auto-solve features using the **Map Coloring (Graph Coloring)** algorithm.

**GOAL**

* To visualize the **Map Coloring algorithm** interactively.
* To implement **real-time constraint checking** during coloring.
* To include **AI-based hint and auto-solve** options using **backtracking search**.
* To allow different **graph structures** and difficulty levels through color limits.

**Expected Result:**  
An interactive Python application where users can experiment with the Map Coloring algorithm visually, gaining an understanding of constraint satisfaction and backtracking search.

**THEORETICAL BACKGROUND**

The **Map Coloring Problem** involves assigning colors to regions or nodes of a graph so that adjacent regions have different colors. Mathematically, it can be represented as a **Graph Coloring** problem — where each node represents a region and edges represent adjacency.

**Key Concepts:**

* **Graph Theory:** Represents maps as vertices (nodes) and edges (connections).
* **Constraint Satisfaction Problem (CSP):** Each node’s color must satisfy constraints with neighbors.
* **Backtracking Algorithm:** Tries color assignments recursively and backtracks when constraints are violated.
* **Four-Color Theorem:** States that any planar map can be colored using only four colors.

**Applications:**  
Map generation, register allocation in compilers, frequency assignment in networks, and game logic design.

**ALGORITHM EXPLANATION WITH EXAMPLE**

Algorithm Used: Backtracking-Based Map Coloring Algorithm

Steps:

1. Choose a node and assign a color from the available color list.
2. Check if any adjacent node has the same color.
3. If valid, move to the next node; else, backtrack and try another color.
4. Repeat until all nodes are colored or no solution exists.

Example:  
For a triangle (A-B-C), if A = Red, B = Green, then C cannot be Red or Green, so assign Blue.  
This continues recursively for larger graphs like squares, houses, or hexagons.

**IMPLEMENTATION AND CODE**

import tkinter as tk

from tkinter import messagebox, ttk

import random

# ---------- Graph definitions ----------

SAMPLE\_GRAPHS = {

"Triangle": {

"nodes": {"A": (200, 100), "B": (100, 250), "C": (300, 250)},

"edges": [("A", "B"), ("B", "C"), ("C", "A")]

},

"Square": {

"nodes": {"A": (150, 100), "B": (300, 100), "C": (300, 250), "D": (150, 250)},

"edges": [("A","B"), ("B","C"), ("C","D"), ("D","A")]

},

"House": {

"nodes": {"A": (150,250), "B": (300,250), "C": (300,100), "D": (150,100), "E": (225,30)},

"edges": [("A","B"),("B","C"),("C","D"),("D","A"),("C","E"),("D","E")]

},

"Map-Like": {

"nodes": {

"A": (150,100), "B": (300,100), "C": (450,150),

"D": (150,250), "E": (300,250), "F": (450,300)

},

"edges": [("A","B"),("B","C"),("A","D"),("B","E"),("C","F"),

("D","E"),("E","F"),("A","E"),("B","D")]

},

"Star": {

"nodes": {

"A": (250,60), "B": (350,150), "C": (300,270),

"D": (200,270), "E": (150,150)

},

"edges": [("A","B"),("A","C"),("A","D"),("A","E"),("B","C"),("C","D"),("D","E"),("E","B")]

},

"Hexagon": {

"nodes": {

"A": (200,80), "B": (300,80), "C": (350,180),

"D": (300,280), "E": (200,280), "F": (150,180)

},

"edges": [("A","B"),("B","C"),("C","D"),("D","E"),("E","F"),("F","A")]

}

}

# ---------- Logic functions ----------

def is\_valid\_coloring(edges, coloring):

for a, b in edges:

if coloring.get(a) == coloring.get(b) and coloring.get(a) is not None:

return False

return True

def valid\_color\_for\_node(edges, node, color, coloring):

for a, b in edges:

if node in (a, b):

neighbor = b if a == node else a

if coloring.get(neighbor) == color:

return False

return True

def backtrack(edges, nodes, colors, coloring, idx=0):

if idx == len(nodes):

return coloring.copy()

node = nodes[idx]

for c in colors:

if valid\_color\_for\_node(edges, node, c, coloring):

coloring[node] = c

res = backtrack(edges, nodes, colors, coloring, idx+1)

if res:

return res

coloring[node] = None

return None

# ---------- GUI Game ----------

class GraphColoringGame:

def \_\_init\_\_(self, root):

self.root = root

self.root.title("Graph Coloring Puzzle Game")

# Layout

self.top\_frame = tk.Frame(root)

self.top\_frame.pack(pady=5)

tk.Label(self.top\_frame, text="Select Shape: ", font=("Arial", 12)).pack(side="left")

self.graph\_choice = ttk.Combobox(self.top\_frame, values=list(SAMPLE\_GRAPHS.keys()) + ["Random"], width=12)

self.graph\_choice.set("Triangle")

self.graph\_choice.pack(side="left", padx=5)

tk.Button(self.top\_frame, text="Load", command=self.load\_graph).pack(side="left", padx=5)

tk.Label(self.top\_frame, text="Colors: ").pack(side="left")

self.color\_count = tk.Spinbox(self.top\_frame, from\_=2, to=5, width=5)

self.color\_count.pack(side="left", padx=5)

self.canvas = tk.Canvas(root, width=600, height=400, bg="white")

self.canvas.pack(pady=10)

self.control\_frame = tk.Frame(root)

self.control\_frame.pack()

tk.Button(self.control\_frame, text="Hint", command=self.show\_hint).pack(side="left", padx=5)

tk.Button(self.control\_frame, text="Solve", command=self.solve).pack(side="left", padx=5)

tk.Button(self.control\_frame, text="Reset", command=self.reset).pack(side="left", padx=5)

# Colors

self.palette = ["#f54242", "#42f554", "#4287f5", "#f5e642", "#f542dd"]

self.graph\_data = None

self.coloring = {}

self.node\_radius = 25

self.load\_graph()

def load\_graph(self):

"""Load selected puzzle."""

name = self.graph\_choice.get()

if name == "Random":

name, graph = random.choice(list(SAMPLE\_GRAPHS.items()))

self.graph\_choice.set(name)

else:

graph = SAMPLE\_GRAPHS.get(name, SAMPLE\_GRAPHS["Triangle"])

self.graph\_data = graph

color\_count = int(self.color\_count.get())

self.colors = self.palette[:color\_count]

self.nodes = list(graph["nodes"].keys())

self.edges = graph["edges"]

self.positions = graph["nodes"]

self.coloring = {n: None for n in self.nodes}

self.draw\_graph()

def draw\_graph(self):

"""Draw nodes and edges."""

self.canvas.delete("all")

# Draw edges

for a, b in self.edges:

x1, y1 = self.positions[a]

x2, y2 = self.positions[b]

self.canvas.create\_line(x1, y1, x2, y2, width=2)

# Draw nodes

for node, (x, y) in self.positions.items():

color = self.coloring[node]

fill = color if color else "lightgray"

outline = "red" if not self.is\_node\_valid(node) else "black"

self.canvas.create\_oval(

x - self.node\_radius, y - self.node\_radius,

x + self.node\_radius, y + self.node\_radius,

fill=fill, outline=outline, width=2, tags=("node", node)

)

self.canvas.create\_text(x, y, text=node, font=("Arial", 12, "bold"))

# Bind clicks

self.canvas.tag\_bind("node", "<Button-1>", self.on\_click)

def on\_click(self, event):

"""Handle node click."""

clicked\_node = None

for node, (x, y) in self.positions.items():

if (x - self.node\_radius <= event.x <= x + self.node\_radius and

y - self.node\_radius <= event.y <= y + self.node\_radius):

clicked\_node = node

break

if not clicked\_node:

return

current = self.coloring[clicked\_node]

if current is None:

self.coloring[clicked\_node] = self.colors[0]

else:

idx = self.colors.index(current)

self.coloring[clicked\_node] = self.colors[(idx + 1) % len(self.colors)]

self.draw\_graph()

if self.check\_complete():

messagebox.showinfo("Congrats!", "🎉 All nodes colored correctly!")

def is\_node\_valid(self, node):

"""Check conflicts."""

color = self.coloring[node]

if not color:

return True

for a, b in self.edges:

if node in (a, b):

neighbor = b if a == node else a

if self.coloring.get(neighbor) == color:

return False

return True

def check\_complete(self):

if None in self.coloring.values():

return False

return is\_valid\_coloring(self.edges, self.coloring)

def show\_hint(self):

"""Suggest one color."""

solution = backtrack(self.edges, self.nodes, self.colors, self.coloring.copy())

if not solution:

messagebox.showwarning("Hint", "No valid solution from current state.")

return

for node in self.nodes:

if self.coloring[node] is None:

color = solution[node]

messagebox.showinfo("Hint", f"Try coloring {node} with {color}.")

return

messagebox.showinfo("Hint", "All nodes already colored!")

def solve(self):

"""Auto-solve puzzle."""

solution = backtrack(self.edges, self.nodes, self.colors, self.coloring.copy())

if not solution:

messagebox.showerror("Solve", "No valid solution exists.")

else:

self.coloring = solution

self.draw\_graph()

messagebox.showinfo("Solved", "✅ Puzzle solved!")

def reset(self):

for n in self.coloring:

self.coloring[n] = None

self.draw\_graph()

# ---------- Run ----------

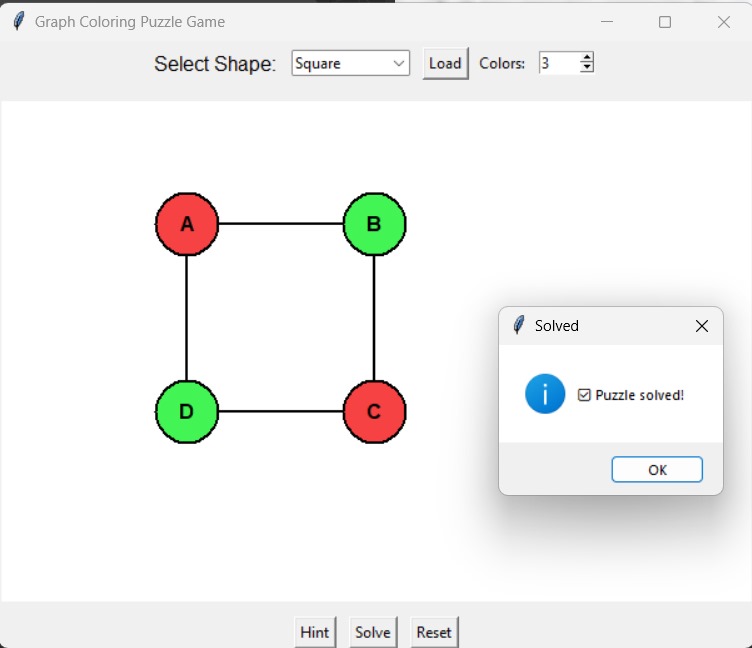
if \_\_name\_\_ == "\_\_main\_\_":

root = tk.Tk()

app = GraphColoringGame(root)

root.mainloop()

**OUTPUT**



**Explanation:**  
The system continuously validates user actions and visually highlights conflicts. When the puzzle is solved correctly, it confirms completion.

**RESULTS AND FUTURE ENHANCEMENT**

**Results:**

* The game successfully demonstrates the map coloring algorithm visually.
* It validates user inputs in real time and provides hints or full solutions.
* It supports multiple graph structures and adjustable color sets.

**Future Enhancements:**

* Add random graph generation and difficulty levels.
* Include animation or scoring based on user performance.
* Extend to **geographical maps** using actual regional data.
* Integrate **machine learning** to predict optimal color sequences.

|  |  |
| --- | --- |
| **Git Hub Link of the project and report** | <https://github.com/arthi13072004-arch/ai_miniproject> |

**REFERENCES**

1. Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach*. Pearson.
2. GeeksforGeeks – *Backtracking | M-Coloring Problem*.
3. TutorialsPoint – *Constraint Satisfaction Problems in AI*.
4. Python Docs – *Tkinter GUI Programming*.
5. Wikipedia – *Four Color Theorem and Graph Coloring*.