**AI Map Coloring**

**1. INTRODUCTION:**

The Map Coloring Problem is a foundational challenge in the fields of graph theory, computer science, and artificial intelligence. It serves as a classic example of a Constraint Satisfaction Problem (CSP), where the objective is to find a state that adheres to a given set of rules or constraints.

This project, the "AI Map Coloring Visualizer," is an interactive web application built with React, TypeScript, and the Google Gemini API. It is designed not only to solve the map coloring problem but also to provide a clear, step-by-step visualization of the process. By allowing users to select different maps, adjust the number of colors, and even upload their own map definitions, the application serves as an engaging educational tool for understanding complex AI search algorithms and heuristics.

**2. PROBLEM STATEMENT**

The problem is to assign a color to each region on a map such that no two regions sharing a common border are assigned the same color. The core challenge is to achieve this using a predefined, and often minimal, number of colors.

This can be formally defined as a Constraint Satisfaction Problem (CSP) with the following components:

* **Variables:** Each region of the map.
* **Domain:** The set of available colors that can be assigned to each region.
* **Constraints:** If two regions are adjacent (share a border), they cannot be assigned the same color.

The application's task is to find a valid assignment of colors to all regions that satisfies all constraints for a given number of colors.

**3. GOAL**

The primary goal of this project is to create a robust, interactive, and educational web application that:

1. **Visualizes the Solution:** Clearly demonstrates the step-by-step process of a backtracking algorithm as it solves the map coloring problem.
2. **Offers User Interactivity:** Allows users to select from pre-defined maps (e.g., Australia, USA West), upload custom maps in JSON format, adjust the number of available colors, and customize the color palette.
3. **Implements an Efficient AI Algorithm:** Uses a backtracking algorithm enhanced with the Minimum Remaining Values (MRV) heuristic to find solutions efficiently.
4. **Provides Educational Content:** Integrates with the Google Gemini API to dynamically fetch and display a clear, well-structured explanation of the underlying theoretical concepts.
5. **Delivers a High-Quality User Experience:** Features a modern, responsive user interface with clear status updates and intuitive controls.

**4. THEORETICAL BACKGROUND**

The solution is built upon several key computer science concepts:

* **Graph Theory:** A map is abstractly represented as a graph. Each region becomes a **node** (or vertex), and an **edge** is drawn between two nodes if their corresponding regions share a border. The map coloring problem is thus transformed into a graph coloring problem.
* **Constraint Satisfaction Problem (CSP):** As previously stated, the problem fits the CSP framework perfectly. The algorithm's job is to find a complete and consistent assignment for all variables.
* **Backtracking Algorithm:** This is a depth-first search algorithm used for solving CSPs. It works by incrementally building a solution and abandoning a path (i.e., "backtracking") as soon as it determines that the path cannot lead to a valid solution. This avoids the computational cost of exploring every possible combination.
* **Minimum Remaining Values (MRV) Heuristic:** To make the backtracking algorithm more efficient, the MRV heuristic is used to decide which node to color next. At each step, the algorithm selects the uncolored node that has the *fewest* legal color choices remaining. This "most constrained variable" strategy often leads to failures and backtracking earlier in the search process, significantly pruning the search tree and speeding up the discovery of a solution.

**5. ALGORITHM EXPLANATION WITH EXAMPLE**

The application uses a backtracking algorithm implemented as a generator function to allow for step-by-step animation. The core logic, incorporating the MRV heuristic, is as follows:

**Algorithm Steps:**

1. **Initialization:** All nodes in the graph are marked as uncolored.
2. **Recursive Solving Loop:**  
   a. **Select Node:** Using the **MRV heuristic**, identify the uncolored node with the fewest valid color options. A valid color is one that is not used by any of its already-colored neighbors. If all nodes are colored, a solution has been found.  
   b. **Iterate Through Colors:** For the selected node, try assigning each available color one by one.  
   c. **Check Constraint:** For a given color, check if it is "safe" (i.e., no adjacent neighbor has this color).  
   d. **Assign and Recurse:** If the color is safe, assign it to the node and make a recursive call to solve for the next node.  
   e. **Backtrack:** If the recursive call returns false (meaning it hit a dead end), or if all colors have been tried for the current node without success, un-assign the current color (reset it to null) and return false. This triggers backtracking in the previous stack frame.
3. **Termination:** The process ends when a complete, valid coloring is found, or when all possibilities have been exhausted, proving no solution exists for the given number of colors.

**Example with the "Simple" Map (3 Colors: Red, Green, Blue):**

* **Nodes:** A, B, C, D, E.
* **Initial State:** All nodes are uncolored.
* **Step 1:** Select Node. The MRV heuristic calculates that node 'B' is the most constrained (most neighbors). Let's select 'B'.
* **Step 2:** Color 'B'. Try **Red**. This is safe. B = Red.
* **Step 3:** Select Next Node. MRV calculates the remaining options. Let's say it selects 'E'.
* **Step 4:** Color 'E'. 'E' is adjacent to 'B' (Red). It can be colored **Green** or **Blue**. Let's try **Green**. E = Green.
* **Step 5:** Select Next Node. Let's say 'D'. 'D' is adjacent to 'B' (Red) and 'E' (Green). The only safe color is **Blue**. D = Blue.
* **Step 6:** Select Next Node. 'A'. 'A' is adjacent to 'B' (Red) and 'D' (Blue). The only safe color is **Green**. A = Green.
* **Step 7:** Select Next Node. 'C'. 'C' is adjacent to 'B' (Red) and 'E' (Green). The only safe color is **Blue**. C = Blue.
* **Result:** All nodes are colored. A valid solution is found: {A:Green, B:Red, C:Blue, D:Blue, E:Green}.

If at any point a node had no safe colors available, the algorithm would backtrack, undoing the previous color assignment and trying a different one.

1. **IMPLEMENTATION AND CODE**

import React, { useState, useEffect, useCallback, useRef } from 'react';

import { ControlsPanel } from './components/ControlsPanel';

import { GraphVisualizer } from './components/GraphVisualizer';

import { InfoPanel } from './components/InfoPanel';

import { MAPS as INITIAL\_MAPS } from './data/maps';

import { solveMapColoringGenerator } from './services/coloringService';

import type { Graph, ColorMapping, Step, MapDefinition } from './types';

import { buildGraph } from './utils/graphUtils';

import { COLOR\_PALETTE as DEFAULT\_PALETTE } from './constants';

const App: React.FC = () => {

const [maps, setMaps] = useState<Record<string, MapDefinition>>(INITIAL\_MAPS);

const [selectedMapKey, setSelectedMapKey] = useState<string>(Object.keys(INITIAL\_MAPS)[0]);

const [numColors, setNumColors] = useState<number>(4);

const [colorPalette, setColorPalette] = useState<string[]>(DEFAULT\_PALETTE.slice(0, 4));

const [currentGraph, setCurrentGraph] = useState<Graph>(buildGraph(maps[selectedMapKey]));

const [colorMapping, setColorMapping] = useState<ColorMapping>({});

const [isSolving, setIsSolving] = useState<boolean>(false);

const [status, setStatus] = useState<string>('Ready. Select a map and click "Color Map" to start.');

const [highlightedNode, setHighlightedNode] = useState<string | null>(null);

const [uploadError, setUploadError] = useState<string | null>(null);

const solverRef = useRef<Generator<Step, ColorMapping | null, void> | null>(null);

const animationFrameId = useRef<number | null>(null);

useEffect(() => {

const mapDef = maps[selectedMapKey];

if (mapDef) {

setCurrentGraph(buildGraph(mapDef));

resetState();

setUploadError(null);

} else if (selectedMapKey === 'custom') {

setCurrentGraph({ nodes: [], edges: [], adjacencyList: {} });

resetState();

setStatus('Please upload a custom map file.');

}

// eslint-disable-next-line react-hooks/exhaustive-deps

}, [selectedMapKey, maps]);

const resetState = useCallback(() => {

if (animationFrameId.current) {

cancelAnimationFrame(animationFrameId.current);

}

solverRef.current = null;

animationFrameId.current = null;

const initialMapping: ColorMapping = {};

const mapDef = maps[selectedMapKey];

if(mapDef) {

mapDef.nodes.forEach(node => {

initialMapping[node.id] = null;

});

}

setColorMapping(initialMapping);

setIsSolving(false);

if (selectedMapKey !== 'custom' || maps.custom) {

setStatus('Ready. Select a map and click "Color Map" to start.');

}

setHighlightedNode(null);

}, [selectedMapKey, maps]);

const handleNumColorsChange = (value: number) => {

setNumColors(value);

// Adjust the palette based on the new number of colors

setColorPalette(prevPalette => {

const newPalette = [...prevPalette];

if (value > newPalette.length) {

// Add new colors from the default palette if numColors increases

const diff = value - newPalette.length;

const additionalColors = DEFAULT\_PALETTE.slice(newPalette.length, newPalette.length + diff);

// Ensure we have fallback colors if default palette is exhausted

for (let i = 0; i < diff - additionalColors.length; i++) {

additionalColors.push(DEFAULT\_PALETTE[i % DEFAULT\_PALETTE.length]);

}

return [...newPalette, ...additionalColors];

} else if (value < newPalette.length) {

// Truncate palette if numColors decreases

return newPalette.slice(0, value);

}

return prevPalette;

});

};

const handleCustomMapUpload = (newMapDef: MapDefinition) => {

const newMaps = { ...maps, custom: newMapDef };

setMaps(newMaps);

setSelectedMapKey('custom');

setStatus('Custom map loaded. Ready to solve.');

setUploadError(null);

};

const handleUploadError = (error: string) => {

setUploadError(error);

};

const animateSolution = useCallback(() => {

if (!solverRef.current) return;

const mapDef = maps[selectedMapKey];

// Guard against race conditions: If the map changes while solving, stop the animation.

if (!mapDef) {

console.error("Solver running without a valid map definition. Stopping.");

resetState();

return;

}

const next = solverRef.current.next();

if (!next.done) {

const { type, nodeId, mapping } = next.value;

const nodeLabel = mapDef.nodes.find(n => n.id === nodeId)?.label ?? nodeId;

setColorMapping(mapping);

setHighlightedNode(nodeId);

switch (type) {

case 'TRY':

setStatus(`Trying to color ${nodeLabel}...`);

break;

case 'SUCCESS':

setStatus(`Successfully colored ${nodeLabel}.`);

break;

case 'BACKTRACK':

setStatus(`Backtracking from ${nodeLabel}...`);

break;

}

animationFrameId.current = requestAnimationFrame(animateSolution);

} else {

setIsSolving(false);

setHighlightedNode(null);

if (next.value) {

setColorMapping(next.value);

setStatus(`Solved successfully with ${numColors} colors!`);

} else {

setStatus(`No solution found with ${numColors} colors. Try increasing the number of colors.`);

}

}

}, [numColors, selectedMapKey, maps, resetState]);

const handleSolve = useCallback(() => {

if (isSolving || !maps[selectedMapKey]) return;

resetState();

setIsSolving(true);

setStatus('Starting solver...');

const graphToSolve = buildGraph(maps[selectedMapKey]);

solverRef.current = solveMapColoringGenerator(graphToSolve, numColors);

animationFrameId.current = requestAnimationFrame(animateSolution);

}, [isSolving, resetState, selectedMapKey, numColors, animateSolution, maps]);

return (

<div className="min-h-screen bg-gray-900 text-gray-100 font-sans p-4 lg:p-8 flex flex-col">

<header className="text-center mb-6">

<h1 className="text-4xl md:text-5xl font-bold text-cyan-400">

<i className="fas fa-palette mr-3"></i>AI Map Coloring

</h1>

<p className="text-gray-400 mt-2 text-lg">Visualizing the Graph Coloring problem with a backtracking algorithm.</p>

</header>

<main className="flex-grow flex flex-col lg:flex-row gap-8">

<div className="lg:w-1/3 xl:w-1/4 bg-gray-800/50 p-6 rounded-2xl shadow-lg border border-gray-700">

<ControlsPanel

maps={maps}

selectedMapKey={selectedMapKey}

onMapChange={setSelectedMapKey}

numColors={numColors}

onNumColorsChange={handleNumColorsChange}

onSolve={handleSolve}

onReset={resetState}

isSolving={isSolving}

onCustomMapUpload={handleCustomMapUpload}

onUploadError={handleUploadError}

uploadError={uploadError}

colorPalette={colorPalette}

onColorPaletteChange={setColorPalette}

/>

<hr className="my-6 border-gray-600"/>

<InfoPanel status={status} />

</div>

<div className="lg:w-2/3 xl:w-3/4 flex-grow bg-gray-800/50 p-4 rounded-2xl shadow-lg border border-gray-700 flex items-center justify-center">

<GraphVisualizer

graph={currentGraph}

colorMapping={colorMapping}

highlightedNode={highlightedNode}

colorPalette={colorPalette}

/>

</div>

</main>

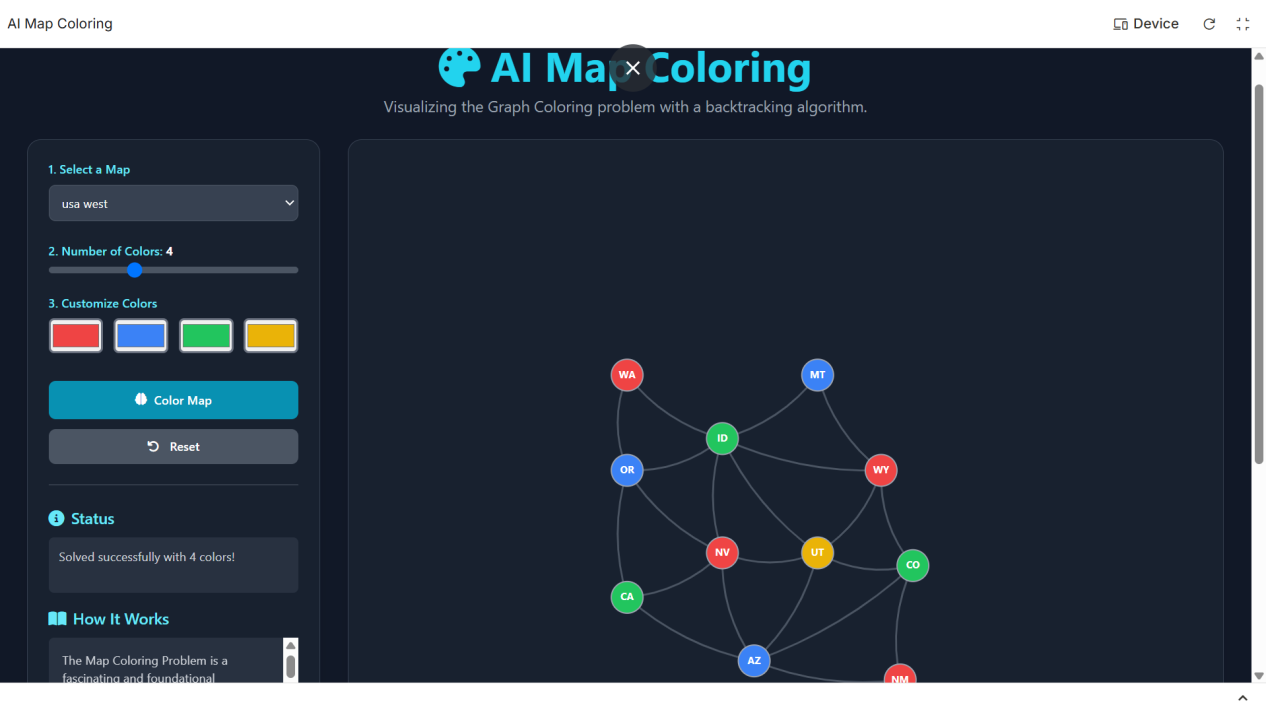
</div>

);

};

export default App;

**7. OUTPUT**



**8. RESULTS AND FUTURE ENHANCEMENT**

**Results:**  
The application successfully implements and visualizes a backtracking solver for the map coloring problem. It correctly finds valid colorings for various maps and adheres to user-defined constraints like the number of colors. The MRV heuristic proves effective in speeding up the search on more complex graphs. The user interface is intuitive, and the integration of the Gemini API adds significant educational value.

**Future Enhancements:**

* **Additional Heuristics:** Implement and allow users to toggle between different heuristics, such as the **Degree Heuristic** (prioritize nodes with the most neighbors) or the **Least Constraining Value** heuristic (choose a color that leaves the most options for neighbors), to compare their performance.
* **Alternative Algorithms:** Integrate other AI algorithms like **Genetic Algorithms** or **Simulated Annealing** to offer a comparative study of different approaches to solving CSPs.
* **Performance Metrics:** Display statistics after a solve is complete, such as execution time, number of nodes visited, and total backtracks, to provide a quantitative measure of algorithm efficiency.
* **Interactive Map Editor:** Develop an in-browser tool that allows users to create their own maps by clicking to add nodes and dragging between them to create edges.
* **Real-World Map Integration:** Use a library like D3.js or a mapping API to allow coloring of actual geographic maps (e.g., the states of India, the countries of Europe) from GeoJSON data.

|  |  |
| --- | --- |
| **Git Hub Link of the project and report** | **https://github.com/deepika-08062007/Map-coloring-AI-mini-project-** |

**9.REFERENCE:**

**Wikipedia: Graph Coloring**

* **Link:** [https://en.wikipedia.org/wiki/Graph\_coloring](https://www.google.com/url?sa=E&q=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FGraph_coloring)

**GeeksforGeeks: M-Coloring Problem & Backtracking**

* **Link:** [https://www.geeksforgeeks.org/m-coloring-problem-backtracking-5/](https://www.google.com/url?sa=E&q=https%3A%2F%2Fwww.geeksforgeeks.org%2Fm-coloring-problem-backtracking-5%2F)

**Brilliant.org: Graph Coloring**

* **Link:** [https://brilliant.org/wiki/graph-coloring/](https://www.google.com/url?sa=E&q=https%3A%2F%2Fbrilliant.org%2Fwiki%2Fgraph-coloring%2F)

**Tailwind CSS Documentation**

* **Link:** [https://tailwindcss.com/docs](https://www.google.com/url?sa=E&q=https%3A%2F%2Ftailwindcss.com%2Fdocs)