Colors - Group Report

Susanna Riccardi, Costanza Volpini, Marco Tollini, Patrick Balestra

The Problem

The problem that we have chosen to solve is defined as follows: given an undirected graph *G = (V, E)* and an integer *k*, is there a way to color the vertices with *k* colors such that adjacent vertices are colored differently?
This problem can be applied in many different fields such as printing a map with the smallest amount of colors, for example.

Encoding

Our solution is completely parameterized. At the top of the Python file, we have three lists that contain the available colors, the links between the nodes and the nodes themselves.
We decided to encode the problem in two steps:

1. Create the variables for the SAT problem.
2. Encode the constraints for the SAT problem.

In the *gen_vars* function we generate all the variables given the k colors and the nodes list. Each color and node is identified with a number between 0 and *n - 1*, where *n* is the total count of nodes or colors in the graph.
We now create a mapping between each color and node to create all the possible pairs. The generated map is then passed to the *genGraphConstr* function which receives as well the count of colors, nodes and links to compute the constraints.
We initialize an empty array named *clauses* which stores all the clauses of the problem that the SAT solver will need. Our problem is very similar to the pigeon hole example code and our implementation can be described as follows:

- We model the constraint to assign exactly one color to each node in the following way: for each node we create a conjunction with all the color pairs for that node. This means that each node may be assigned 0 or 1 color. To enforce the uniqueness of having only one color, we add a single clause that ORs all the colors for that node. As an example, we can consider the colors red (r), green (g) and blue (b) to be assigned to node 1. We can encode this constraint in CNF format as follows:

$(\neg 1r \vee \neg 1b) \wedge (\neg 1r \vee \neg 1g) \wedge (\neg 1b \vee \neg 1g) \wedge (1r \vee 1g \vee 1b)$

- We model the constraint to assign a color to a node with one of the available colors that is not yet taken by one of its neighbors in the following way: for each color and node, we find the neighbors of the node and encode that at most one node has that color. As an example, we can consider that node 1 is connected to node 2, 3 and 4 with the same available colors as before. We can encode this constraint in CNF format as follows:
  $(\neg 1r \vee \neg 2r) \wedge (\neg 1r \vee \neg 3r) \wedge (\neg 1r \vee \neg 4r)$

We use the *json* Python package to load the file *graph.json* which contains the graph encoded in a dictionary with the colors, nodes and links. The SAT solver will attempt to return the solution that can satisfy the constraints or simply say that it's unsatisfiable. A new file *solution.json* is written containing whether it's satisfiable or not and if solvable, the pairs that constitute the solution.

Architecture

We have decided to create a static website that the user can interact with. A graph can be created and the user selects the number of colors they would like to assign to the nodes.

The backend server uses Node.js to serve a static website that is written in React. This single-page app allows the user to create nodes and edge interactively. If the user doesn't want to spend time in creating the graph manually, we have developed the possibility to generate a random graph with a given number of nodes. The number of colors is also required to start the computation. The colors are generated randomly and evenly to be able to distinguish the nodes easily.

When the user decides to start the computation, a POST request is sent to the */problem* endpoint with a JSON file that encodes the graph. The data is then written by the server in the solver directory which also launches a child process using the *exec* command to run the Python solver. The resulting file is then read and returned as response to color the graph of the correct colors or display the unsolvability of the problem.

User Interface

We have tried to keep the user interface as simple as possible. We have used the **vis library** (https://github.com/almende/vis) which is a dynamic visualization library to interactively draw a graph. The user can draw to add nodes and connect them with edge by moving their mouse or let our website generate a random graph given a number of nodes. Pressing the "SAT Solver" button will ask for the number of colors to try and use as node colors. In case the problem is satisfiable, the graph is colored with one of the solution returned by the solver. Otherwise, the website displays an error to communicate the unsolvability of the problem with the given constraints.

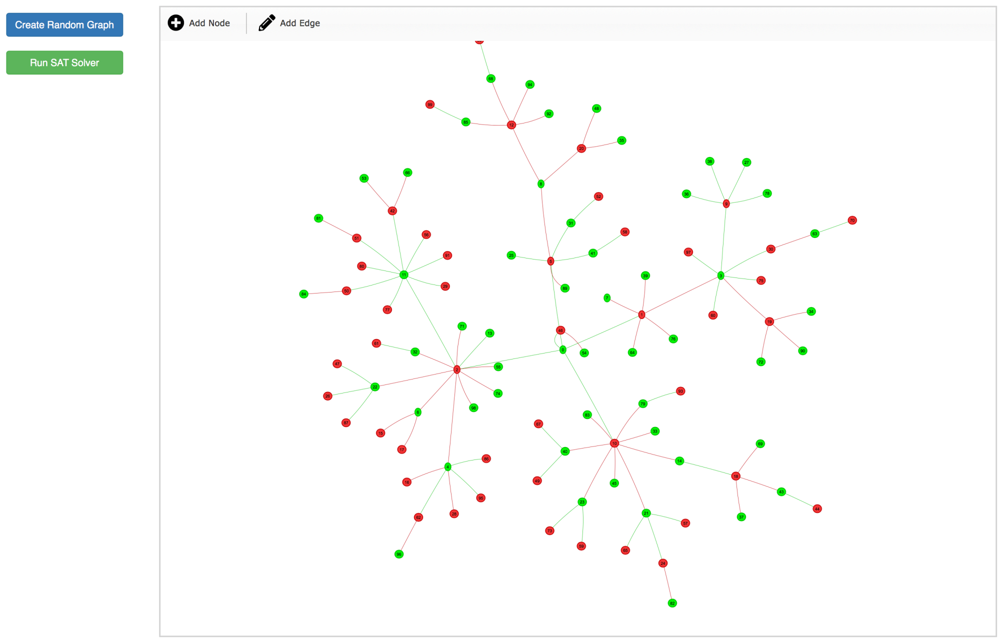

Figure 1: a simply randomly generated graph is colored with 2 colors.

Problems Encountered

Creating a random graph with many nodes can sometime require a few seconds to generate due to the way our algorithm works. Also, as it could be expected, running the solver with many nodes and many colors will slow down the solver.
In the first stages of the project, we also encountered some issues in the positioning of the graph. Once the graph is randomly generated, the page didn't store the position of the nodes, and applying some changes to other React components caused the graph to reload. To fix this issue, we used a different approach to workaround this issue in the library that we use.