Theory of Computation: Graph Coloring Project Report

Enrico Benedettini Giorgio Bonetto Mohamed Ali Atwi
Riccardo Carmellini

May 20, 2024

1 Introduction

This report details the development and implementation of a solution to the graph coloring problem, undertaken as part of the Theory of Computation course at USI (Università della Svizzera italiana). The objective was to determine if a given undirected graph can be colored using k colors such that no two adjacent vertices share the same color. The project was divided into four main steps:

1. Encode Graph Input to Variables
2. Create Propositional Logic from Variables
3. Decode Solution from SAT Solver
4. Create a UI with Vue.js

2 Project Structure

The project consists of a backend implemented in Python, which handles the encoding of the graph coloring problem into SAT, and a frontend developed using Vue.js for user interaction and visualization.

2.1 Backend

The backend comprises two main scripts:

- `sat_solver.py`: Handles the encoding of the graph into SAT, interfacing with the SAT solver, and decoding the solution.
- `backend.py`: A Flask server that processes user input, invokes the solver, and returns the solution.

2.2 Frontend

The frontend is a Vue.js application, allowing users to input graph data, submit it to the backend, and visualize the results.

3 Step-by-Step Development

3.1 Encode Graph Input to Variables

The graph is read from an input file, and variables representing each node-color combination are generated. The function `read_graph` reads the graph structure from a file and initializes global variables. The function `gen_vars` generates variables for each node-color combination.

Listing 1: Graph Input Reading

```
def read_graph(file_name):
    global k, graph, nodes_list, node_to_index
    raw_graph = open(file_name, "r").read()
    raw_graph = raw_graph.split("\n")
    if len(raw_graph[0].split()) != 2:
        print("wrong header for input file : <number of nodes> <number of c
        sys.exit(1)
    k = int(raw_graph[0].split()[1])
    raw_graph = raw_graph[1:]
    for line in raw_graph:
        if not line:
            continue
        line = line.split()
        n = str(line[0])
        if n in graph:
            graph[n].update(line[1:])
        else:
            graph[n] = set(line[1:])
        for v in line[1:]:
            v = str(v)
            if v in graph:
                graph[v].add(line[0])
            else:
                graph[v] = set(line[0])

    nodes_list = list(graph.keys())
    nodes_list.sort()
    for n in range(len(nodes_list)):
        node_to_index[nodes_list[n]] = n

def gen_vars():
    varMap = {}
    for p in range(len(graph.keys())):
        for c in range(k):
            vs = "P%d_k%d" % (p, c)
            varMap[vs] = gvi(vs)
    return varMap
```

3.2 Create Propositional Logic from Variables

The `generate_constraint` function creates the clauses for the SAT solver. This is a crucial function as it contains most of the project's logic. The representation of the problem is given by three clauses:

- each node is assigned at least one color
- each node is assigned no more than one color
- adjacent nodes have different colors

Listing 2: Constraint Generation

```
def generate_constraint(vars):
    clauses = []
    for p in range(len(nodes_list)):
        node_colors = []
        for c in range(k):
            node_colors.append(vars["P%d_k%d" % (p, c)])
        clauses.append(node_colors)

    for p in range(len(nodes_list)):
        for c_i in range(k):
            for c_j in range(c_i + 1, k):
                clauses.append([-vars["P%d_k%d" % (p, c_i)],
                -vars["P%d_k%d" % (p, c_j)]])

    for p_i in range(len(nodes_list)):
        for n in graph[nodes_list[p_i]]:
            for c_i in range(k):
                clauses.append([-vars["P%d_k%d" % (p_i, c_i)],
                -vars["P%d_k%d" % (node_to_index[n], c_i)]])
    return clauses
```

3.2.1 Challenges and Solutions

One of the main challenges we faced was ensuring that the constraints correctly represented the graph coloring problem. Initially, we tried various methods to enforce that each node should have exactly one color and that no two adjacent nodes should share the same color. We realized that adding separate clauses for "at least one color" and "at most one color" constraints for each node was crucial.

To generate these constraints efficiently, we constructed clauses iteratively, ensuring that each node-color combination was properly handled. This approach helped reduce the overall complexity of the constraints and made the SAT solver's task very simple to manage.

3.3 Decode Solution from SAT Solver

The `decode_solution` function parses the SAT solver's output to determine the coloring of the graph or to conclude that no valid coloring exists.

Listing 3: Solution Decoding

```
def decode_solution(output):
    lines = output
    if lines[0] == "s SATISFIABLE":
        solution = {}
        assignment = map(int, lines[1].split()[1:-1])
        for var in assignment:
            if var > 0:
                var_name = varToStr[abs(var)]
                match = re.match(r"P(\d+)_k(\d+)", var_name)
                if match:
                    node_index, color_index = map(int, match.groups())
                    node_name = nodes_list[node_index]
                    solution[node_name] = color_index
        return "satisfiable", solution
    else:
        return "unsatisfiable", {}
```

3.3.1 Challenges and Solutions

Creating an intuitive and responsive user interface was another significant challenge. We experimented with various design patterns and libraries to achieve a balance between usability and functionality. Using Vue.js in combination with D3.js, we were able to create dynamic and interactive visualizations for the graph.

Ensuring smooth integration between the frontend and backend involved configuring CORS and handling asynchronous requests efficiently. By carefully managing the state and responses in the Vue.js application, we were able to provide a seamless user experience.

3.4 Create a UI with Vue.js

A user interface was developed to allow users to input graph data, submit it for solving, and visualize the results.

3.4.1 Frontend-Backend Communication

Setting up communication between the frontend and backend was a key challenge. We utilized Flask-CORS to enable Cross-Origin Resource Sharing (CORS), allowing the Vue.js frontend to make requests to the Flask backend.

Listing 4: Enabling CORS in Flask

```
from flask import Flask, request, jsonify, abort
from flask_cors import CORS

app = Flask(__name__)
CORS(app)
```

Listing 5: Vue.js Axios Configuration

```
fetch('http://127.0.0.1:3000/solve', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(data)
})
.then(response => response.json())
.then(result => {
  this.result = result;
})
.catch(error => {
  console.error('Error:', error);
});
```

3.4.2 Challenges and Solutions

Creating an intuitive and responsive user interface was another significant challenge. We experimented with various design patterns and libraries to achieve a balance between usability and functionality. Using Vue.js in combination with D3.js, we were able to create dynamic and interactive visualizations for the graph.

Ensuring smooth integration between the frontend and backend involved configuring CORS and handling asynchronous requests efficiently. By carefully managing the state and responses in the Vue.js application, we were able to provide a seamless user experience.

4 User Interface

The Vue.js frontend allows users to:

- Enter the number of nodes and edges manually.
- Upload a file containing graph data.
- Specify the number of colors.
- Visualize the initial graph and the resulting coloring.

The Vue.js frontend allows users to interact with the graph coloring problem in a user-friendly manner. Users can input graph data either manually or by uploading a file, specify the number of colors, and visualize the resulting graph coloring.

Functionality

The user interface provides the following functionalities:

- **Manual Input**: Users can manually enter the number of nodes, the edges connecting the nodes, and the number of colors to be used for coloring the graph.

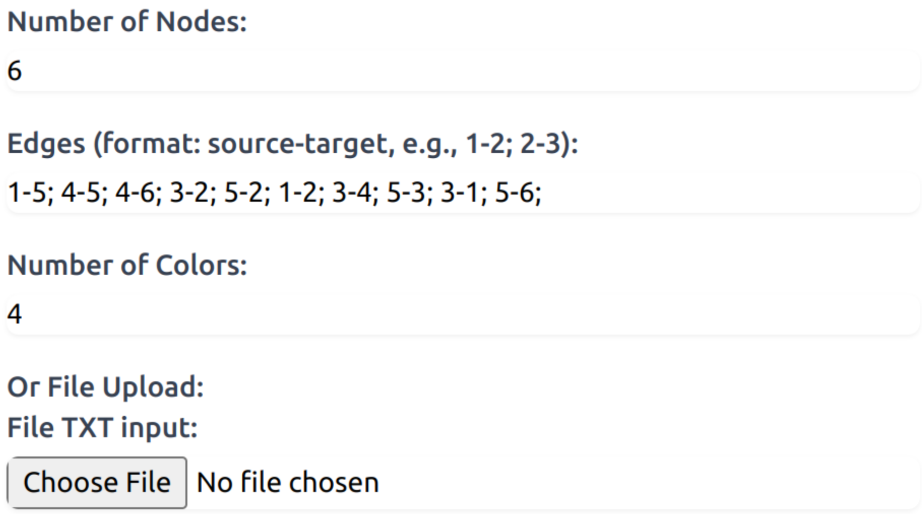

Figure 1: Graph Input Section

- **File Upload**: Users can upload a file containing the graph data in a predefined format. This file should include the number of nodes, the edges, and the number of colors.
- **Solve and Visualize**: After inputting the graph data and specifying the number of colors, users can click a button to solve the graph coloring problem. The backend processes the input, runs the SAT solver, and returns the result, which is then visualized in the frontend.

Possible Scenarios

The project can handle different scenarios based on the input provided by the user:

- **Satisfiable Coloring**: If the graph can be colored using the specified number of colors such that no two adjacent nodes share the same color, the solution is considered satisfiable. The visualization will display the graph with nodes colored accordingly.
- **Unsatisfiable Coloring**: If it is not possible to color the graph with the specified number of colors while satisfying the constraints, the solution is considered unsatisfiable. The visualization will indicate that no valid coloring exists for the given input.

Visualizations

The following images illustrate different aspects of the user interface and the results of the graph coloring problem.

Figure 1 shows the graph input section where users can enter the number of nodes, edges, and the number of colors manually. This section also includes an option for uploading a file containing the graph data.

Figure 2 illustrates the graph visualization when the problem is satisfiable. In this scenario, the SAT solver finds a valid coloring where no two adjacent nodes share the same

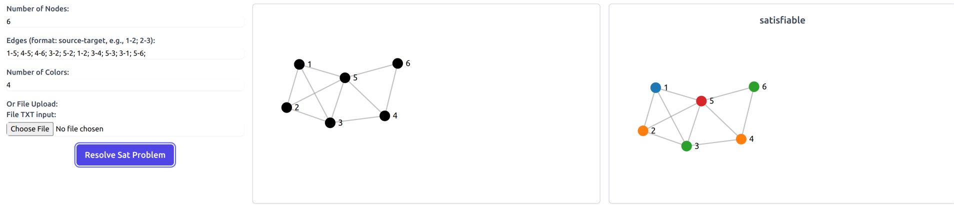

Figure 2: Graph Visualization with Satisfiable Coloring

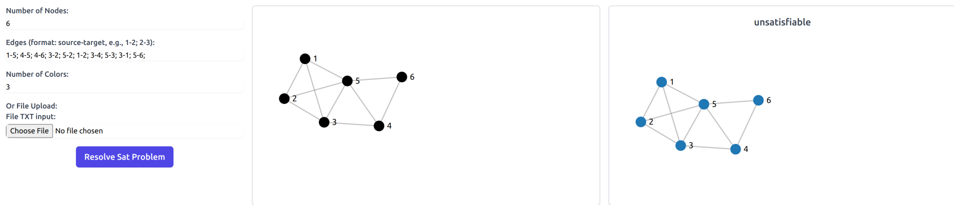

Figure 3: Graph Visualization with Unsatisfiable Coloring

color. The visualization shows the graph with nodes colored according to the solution provided by the SAT solver.

Figure 3 depicts the graph visualization when the problem is unsatisfiable. In this scenario, the SAT solver determines that it is not possible to color the graph with the specified number of colors while satisfying the constraints. The visualization indicates that no valid coloring exists for the given input.

Through these functionalities and visualizations, the project provides an interactive and intuitive way for users to explore and understand the graph coloring problem and its solutions.

5 Conclusion

The project successfully implemented a solution for the graph coloring problem using SAT. The division of tasks ensured a structured approach, resulting in an efficient backend and an intuitive frontend. Future improvements could include optimizing the SAT encoding and enhancing the UI for better user experience.

6 References

- Course materials from the Theory of Computation course at USI.
- Documentation for the SAT solver (e.g., Z3).
- Vue.js and D3.js documentation for frontend development.