CS-470: Artificial Intelligence: Project 3

Taylor Martin
University of Idaho
Moscow, Idaho, USA
mart8517@vandals.uidaho.edu

1 ABSTRACT

This project utilized global search and local search techniques from the field of Artificial Intelligence, specifically from the content covered in CS-470. The main objective was to implement multiple versions of a Constraint Satisfaction Problem (CSP) algorithm to solve a map coloring problem. To accomplish this, two variations of Depth-First Search (DFS) were implemented, incorporating the heuristics "minimum remaining values" and "most constrained variable." Additionally, a local search algorithm employing the "minimum conflicts" heuristic was implemented. The implemented DFS algorithms exhibited satisfactory performance in finding valid colorings with limited colors and quick processing times. However, the local search algorithm encountered difficulties and was unable to find solutions within the given constraints. Further research and improvements are required to enhance the capabilities of the local search algorithm and address the limitations experienced in this project.

2 DESCRIPTION OF THE CSP ALGORITHMS UTILIZED

This section provides a brief description of three different algorithms used for solving Constraint Satisfaction Problems (CSPs): DFS with Minimum Remaining Values (DFS-MRV), DFS with Most Constrained Variable (DFS-MCV), and Local Search with Minimum Conflicts.

2.1 DFS-MRV:

The DFS-MRV algorithm is a exhaustive search algorithm that explores the search space of a CSP using a depth-first traversal strategy. It selects variables to assign values to based on the Minimum Remaining Values (MRV) heuristic, which prioritizes the variables with the fewest remaining legal values. This heuristic aims to reduce the branching factor and increase the likelihood of finding a solution quickly by focusing on the most constrained variables first.

During the search, DFS-MRV produces constraints and performs backtracking when a variable assignment leads to a conflict. It exhaustively explores the search tree, trying different assignments until a valid solution is found or all possibilities have been exhausted.

2.2 DFS-MCV:

Similar to DFS-MRV, the DFS-MCV algorithm is also based on a depth-first traversal strategy. However, it selects variables to assign values to based on the Most Constrained Variable (MCV) heuristic, which prioritizes the variables with the most constraints on other variables. By selecting the most constrained variable, DFS-MCV aims to quickly identify potential conflicts and reduce the overall search space.

As with DFS-MRV, DFS-MCV applies constraint production and backtracking to navigate the search space and find valid solutions. It exhaustively explores the search tree, considering different variable assignments and backtracking when conflicts arise.

2.3 Local Search with Minimum Conflicts:

The Local Search with Minimum Conflicts algorithm takes a different approach to solving CSPs. It is an iterative improvement algorithm that starts with an initial assignment of values to variables and iteratively improves the assignment by minimizing conflicts. Instead of systematically exploring the search space, it focuses on resolving conflicts between variables to reach a valid solution.

At each iteration, the algorithm selects a variable with the highest number of conflicts (the most conflicting variable) and assigns it a value that minimizes conflicts with other variables. This process continues until either a valid solution is found or a termination condition is met (e.g., maximum number of steps or a predefined threshold of conflicts).

Local Search with Minimum Conflicts is particularly useful for CSPs with large search spaces, as it tends to converge towards a solution by iteratively improving the assignment. However, it does not guarantee finding an optimal solution and can sometimes get stuck in local optima.

3 RESULTS

The DFS algorithms demonstrated moderate success in finding valid colorings with a limited number of colors. Initially, the DFS algorithms encountered difficulties in finding a valid solution when restricted to only two colors. However, subsequent iterations of the algorithms successfully identified valid solutions when the number of allowable colors was increased to three or more. The DFS algorithm with the "Fewest Remaining Value" heuristic, when provided with three colors, achieved a process time of 0.00047 seconds. Similarly, the DFS algorithm with the "Most Constrained Variable" heuristic, also with three colors, achieved a process time of 0.00049 seconds. These results indicate the efficiency and effectiveness of DFS algorithms in solving the map coloring problem.

3.1 DFS Valid Solutions

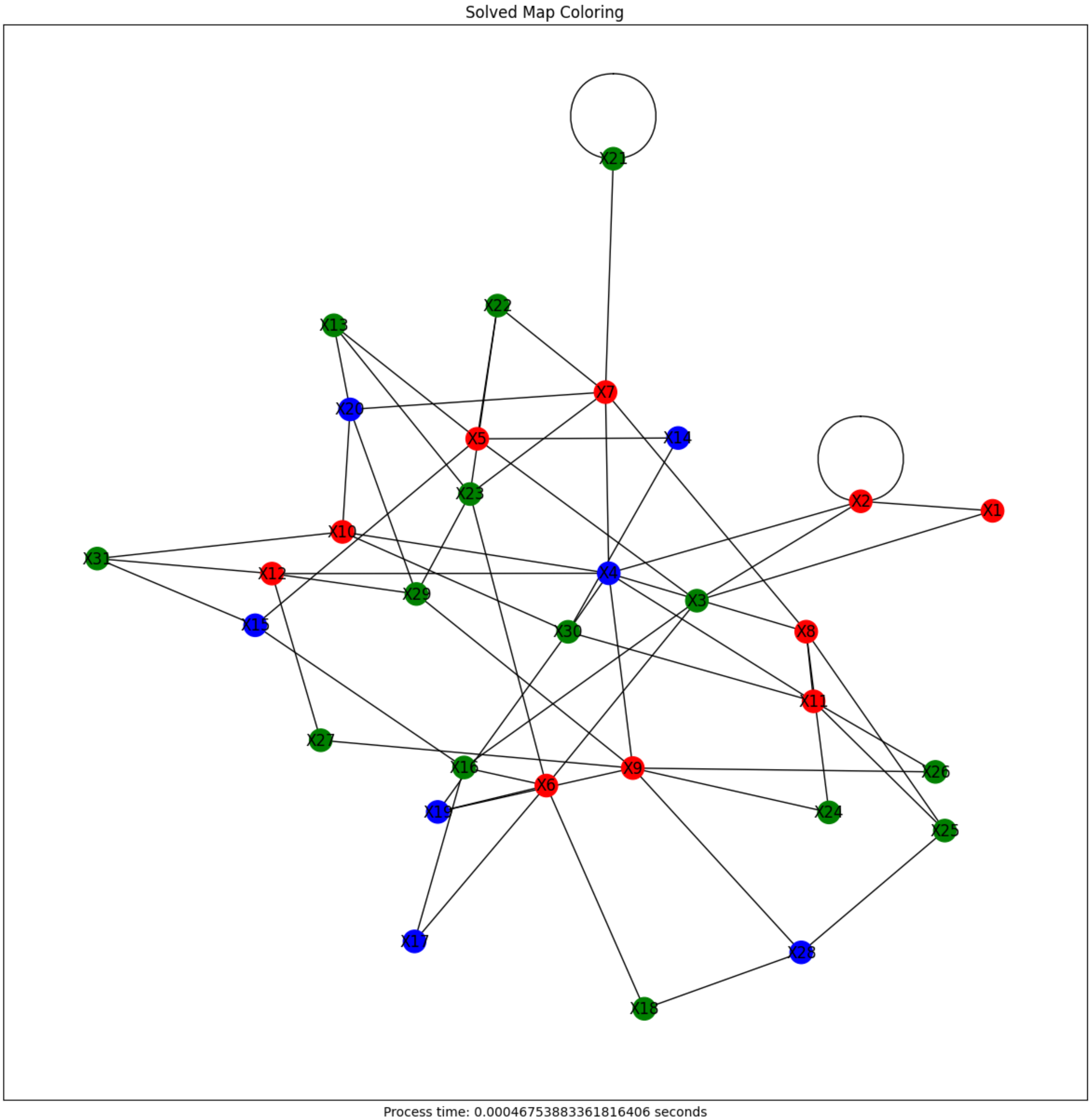

Figure 1: DFS Using Fewest Remaining Value Heuristic With 3 Colors. Process Time: 0.00047

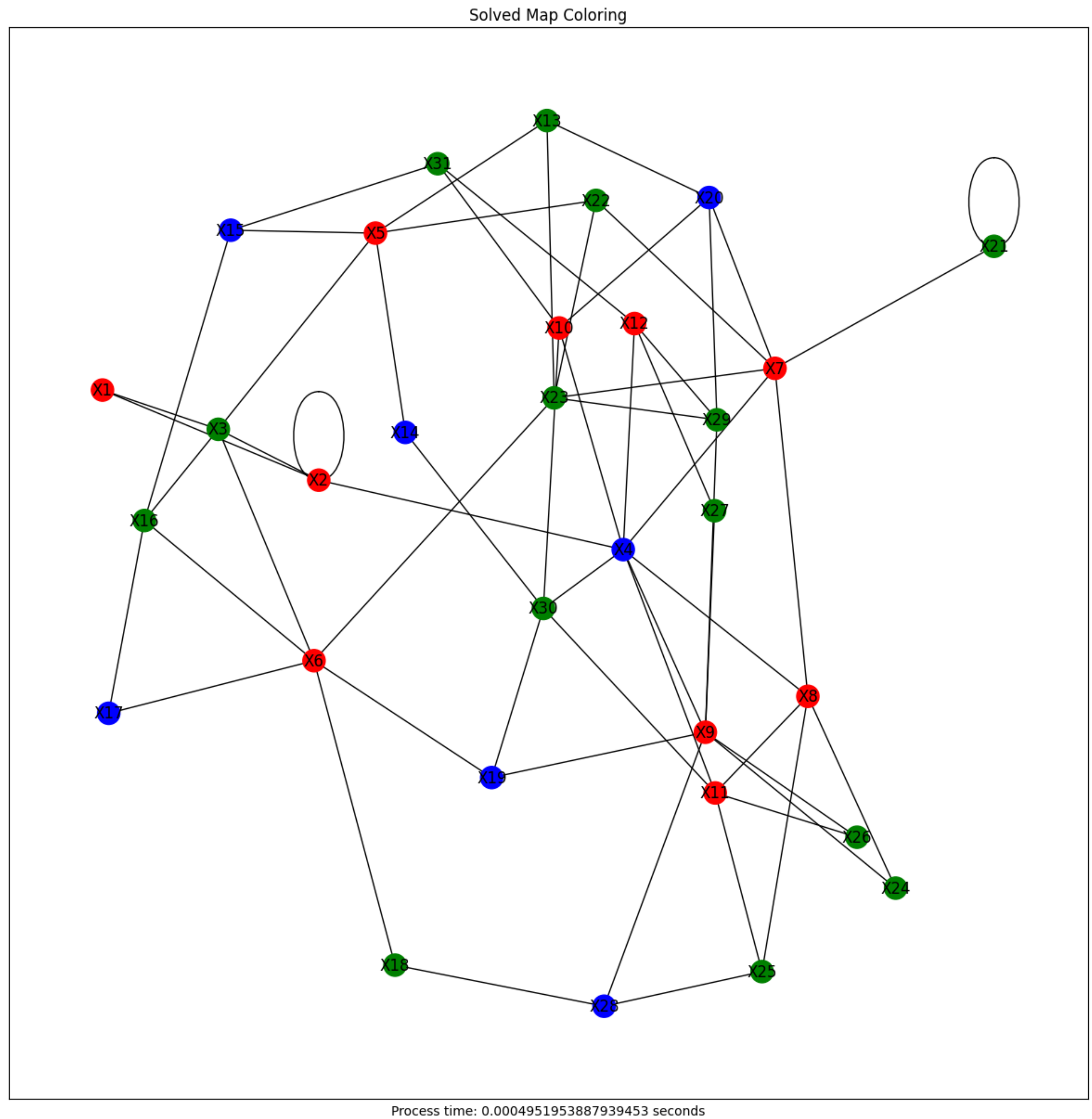

Figure 2: DFS Using Most Constrained Variable Heuristic With 3 Colors. Process Time: 0.00049

3.2 Unsuccessful Attempts in Applying Local Search to Solve the Map Coloring Problem

However, the local search algorithm faced challenges and failed to find a valid coloring in numerous attempts. Various combinations of maximum steps and colors were tested, but the algorithm consistently fell short. For instance, with a maximum of 1,000 steps and 10 colors, the process time was 0.013815879821777344 seconds, while increasing the maximum steps to 1,000,000 resulted in a significantly longer process time of 320.75973558425903 seconds. Similarly, other combinations of maximum colors and steps yielded similar unsuccessful outcomes.

Max Colors	Max Steps	Result	Process Time (sec)
4	1,000	Failed	0.0194
4	10,000	Failed	0.1705
4	100,000	Failed	1.1574
4	1,000,000	Failed	12.9996
10	1,000	Failed	0.0138
10	10,000	Failed	0.1614
10	100,000	Failed	1.3789
10	1,000,000	Failed	315.7407
1,000	100,000	Failed	33.1605
1,000	1,000,000	Failed	320.7597
10,000	100,000	Failed	316.00442
100,000	10,000	Failed	14.5699
100,000	100,000	Failed	3114.9191

4 CONCLUSION

In conclusion, this project successfully utilized global search and local search techniques from the field of Artificial Intelligence, specifically from the content covered in CS-470. The main objective was to implement multiple versions of a Constraint Satisfaction Problem (CSP) algorithm to solve a map coloring problem. To accomplish this, two variations of Depth-First Search (DFS) were implemented, incorporating the heuristics "minimum remaining values" and "most constrained variable." Additionally, a local search algorithm employing the "minimum conflicts" heuristic was implemented. The implemented DFS algorithms exhibited satisfactory performance in finding valid colorings with limited colors and quick processing times. However, the local search algorithm encountered difficulties and was unable to find solutions within the given constraints. Further research and improvements are required to enhance the capabilities of the local search algorithm and address the limitations experienced in this project.

5 CODE APPENDIX

```
# --------------------
# @file    build_graph.py
# @author  Taylor Martin
# @date    June 2023
# @class   CS 470 Artificial Intelligence
# @project Project 3 - CSP
# @brief   This file contains the implementation of a
     function to build a graph from a CSV file. In the
     format of X: [Y, Z, ...] where X is a variable and Y,
     Z, ... are the variables that X is constrained by.
# --------------------

import csv

def parse_csv_file(filename):
    variables = []
    constraints = []

    with open(filename, 'r') as file:
        reader = csv.reader(file)
        data = list(reader)

    num_variables = len(data[0])

    # Create variables
    for i in range(num_variables):
        variable = f'X{i+1}'
        variables.append(variable)

    # Create constraints
    for i in range(num_variables):
        for j in range(i + 1, num_variables):
            if data[i][j] == '1':
                constraint = (variables[i], variables[j])
                constraints.append(constraint)

    #combine variables and constraints into a single
         dictionary
    graph = {}
    for i in range(num_variables):
        graph[variables[i]] = constraints[i]


    return graph


def print_graph(graph):
    """
    Print a graph

    Args:  graph (dict): A dictionary of lists

    Returns:  None

    """
    for variable, neighbors in graph.items():
        print(f"{variable}: {neighbors}")


def build_graph(filename):
    """
    Build a graph from a CSV file. In this case, the CSV
         file is a map of regions that are neighbors. The
         graph is a dictionary of lists. The keys are the
         regions and the values are the neighbors of the
         region.

    Args:  filename (str): The name of the CSV file to parse

    Returns:  None

    Graph example:
```

```python
    {
        'A': ['B', 'C', 'D'],
        'B': ['A', 'C', 'D'],
        'C': ['A', 'B', 'D'],
        'D': ['A', 'B', 'C']
    }

    """
    graph = parse_csv_file(filename)
    print_graph(graph)
    return graph
```

```python
# --------------------
# @file    dfs.py
# @author  Taylor Martin
# @date    June 2023
# @class   CS 470 Artificial Intelligence
# @project Project 3 - CSP
# @brief   This file contains the implementation of the DFS
     algorithm with two different heuristics. Least
     Constraining Value and Most Constrained Variable. To
     solve the map coloring CSP.
# --------------------



def select_most_constrained_variable(graph, colors):

    min_remaining_values = float('inf')
    most_constrained_variable = None

    for node in graph:

        if colors[node] is None:

            remaining_values = len(set(colors[neighbor] for
                 neighbor in graph[node]))

            if remaining_values < min_remaining_values:
                min_remaining_values = remaining_values
                most_constrained_variable = node

    return most_constrained_variable


def dfs_most_constrained_variable(graph, colors):

    if all(colors[node] is not None for node in graph):
        return colors

    node = select_most_constrained_variable(graph, colors)

    for color in ['Red', 'Green', 'Blue']:

        if is_valid_color(node, color, graph, colors):

            colors[node] = color
            result = dfs_most_constrained_variable(graph,
                 colors)

            if result is not None:
                return result

            colors[node] = None

    return None

def is_valid_color(node, color, graph, colors):

    for neighbor in graph[node]:

        if colors[neighbor] == color:

            return False

    return True

def select_fewest_remaining_values_variable(graph, colors):

    min_remaining_values = float('inf')
    fewest_remaining_values_variable = None

    for node in graph:

        if colors[node] is None:

            remaining_values = len(set(colors[neighbor] for
                 neighbor in graph[node]))

            if remaining_values < min_remaining_values:

                min_remaining_values = remaining_values
                fewest_remaining_values_variable = node

    return fewest_remaining_values_variable

def dfs_fewest_remaining_values(graph, colors):

    if all(colors[node] is not None for node in graph):
        return colors

    node = select_fewest_remaining_values_variable(graph,
         colors)

    for color in ['Red', 'Green', 'Blue']:

        if is_valid_color(node, color, graph, colors):

            colors[node] = color

            result = dfs_fewest_remaining_values(graph,
                 colors)

            if result is not None:
                return result

            colors[node] = None

    return None
```

```python
# --------------------
# @file    local_search.py
```

```python
# @author  Taylor Martin
# @date    June 2023
# @class   CS 470 Artificial Intelligence
# @project Project 3 - CSP
# @brief   This file contains the implementation of a local
     search algorithm with the min conflict heuristic. To
     solve the map coloring CSP.
# --------------------


import random

def generate_initial_state(graph, num_colors):

    variables = list(graph.keys())

    state = {}

    for variable in variables:
        state[variable] = random.randint(1, num_colors)

    return state

def count_conflicts(variable, color, state, graph):

    conflicts = 0
    neighbors = graph[variable]

    for neighbor in neighbors:

        if state[neighbor] == color:
            conflicts += 1

    return conflicts

def min_conflicts(graph, num_colors, max_steps):

    state = generate_initial_state(graph, num_colors)

    for _ in range(max_steps):

        conflicted_variables = [variable for variable in
             graph if count_conflicts(variable,
             state[variable], state, graph) > 0]

        if len(conflicted_variables) == 0:
            return state

        variable = random.choice(conflicted_variables)
        min_conflicts = float('inf')
        min_color = None

        for color in range(1, num_colors + 1):

            conflicts = count_conflicts(variable, color,
                 state, graph)

            if conflicts < min_conflicts:
                min_conflicts = conflicts
                min_color = color

        state[variable] = min_color

    return None
```

```python
# --------------------
# @file    create_graphic.py
# @author  Taylor Martin
# @date    June 2023
# @class   CS 470 Artificial Intelligence
# @project Project 3 - CSP
# @brief   This file contains the implementation of a
     function to visualize the map coloring solution.
# --------------------

import networkx as nx
import matplotlib.pyplot as plt

def visualize_map(graph, node_colors, process_time):
    # Create a new figure
    plt.figure(figsize=(15, 15))

    # Create a layout for the nodes
    pos = nx.spring_layout(graph)

    # Draw the graph with node colors
    nx.draw_networkx(graph, pos,
         node_color=list(node_colors.values()))

    # Set axis labels and title
    plt.xlabel(f"Process time: {process_time} seconds")
    plt.title('Solved Map Coloring')

    # Show the plot
    plt.show()
```

```python
# --------------------
# @file    main.py
# @author  Taylor Martin
# @date    June 2023
# @class   CS 470 Artificial Intelligence
# @project Project 3 - CSP
# @brief   This file contains the implementation of the
     main function to run the CSP algorithms.
# --------------------

from build_graph import build_graph
from dfs import dfs_most_constrained_variable ,
     dfs_fewest_remaining_values
from local_search import min_conflicts
from time import time
from create_graphic import visualize_map
import networkx as nx
import matplotlib.pyplot as plt


def show_results(graph, solution, process_time):

    if solution is None:
        print("Failed to find a valid coloring.")
        print(f"Process time: {process_time} seconds")
        return
```

```
    else:
        print("Map coloring solution:")
        for node, color in solution.items():
            print(f"{node}: {color}")
        print(f"Process time: {process_time} seconds")

    nodes = []
    for node in graph:
        nodes.append(node)

    edges = []
    for node in graph:
        for neighbor in graph[node]:
            edges.append((node, neighbor))

    graph = nx.Graph()
    graph.add_nodes_from(nodes)
    graph.add_edges_from(edges)

    # Visualize the graph with colored nodes
    visualize_map(graph, solution)


def main():
    filename = "CSPData.csv"
    graph = build_graph(filename)

    # Initialize colors
    colors = {node: None for node in graph}

    # Running the algorithm
    start_time = time()
    solution = dfs_most_constrained_variable(graph, colors)
    end_time = time()
    process_time = end_time - start_time
    show_results(graph, solution, process_time)

    colors = {node: None for node in graph}

    start_time = time()
    solution2 = dfs_fewest_remaining_values(graph, colors)
    end_time = time()
    process_time = end_time - start_time
    show_results(graph, solution2, process_time)

    start_time = time()
    solution3 = min_conflicts(graph, 1000, 10000)
    end_time = time()
    process_time = end_time - start_time
    show_results(graph, solution3, process_time)


if __name__ == "__main__":
    main()
```