Laboratory 2

Variant 5

Group 5

By Jan Szachno and Aleksandra Głogowska

---

1. Introduction

1.1. Task description

The task was to write a Python program that solves a map coloring problem as a constraint satisfaction problem using backtracking with forward checking. The map is a dictionary in which each region is mapped to the list of its neighbors. The goal is to return the first found solution to the given map coloring problem. The task required writing code, testing it, visualizing the maps, and drawing conclusions after writing the algorithms. The visualizations should be of at least two maps with at least 10 regions, colored so that two adjacent regions are not the same color. Explaining how the domains, constraints, and variables were defined, was also a part of the problem since the task required the use of a given Python template with initializations of the methods, including parameters.

1.2. Algorithm description

The algorithm that will be used to solve the map coloring problem is backtracking with forward checking. Backtracking alone is a technique assuming that if the algorithm stops at a dead end, the next step does not satisfy the constraints of the program - it has to get back to the previous step and change the solution. It can either backtrack one step or multiple steps or even go to the beginning of the variables. In this case, the algorithm tries to solve the map coloring problem, but it can be used in many other standard problems, such as Sudoku, N-Queen, Cryptarithmetic Puzzles, or plan scheduling in schools, universities, and workplaces. In this task, the algorithm starts with an empty assignment of colors to regions. It iteratively selects a region, assigns a color to it, and checks if the assignment violates any constraints (for example, if any neighboring regions have the same color). If a violation is detected, the algorithm backtracks to the previous assignment and tries a different color for the current region.

This algorithm has been enhanced with forward checking. It is used to reduce the search space by eliminating inconsistent values from the domain of unassigned variables. After assigning a color to a region, forward checking updates the domains of neighboring regions by removing the assigned color from their domains. This helps detect potential conflicts early and prune branches of the search tree.

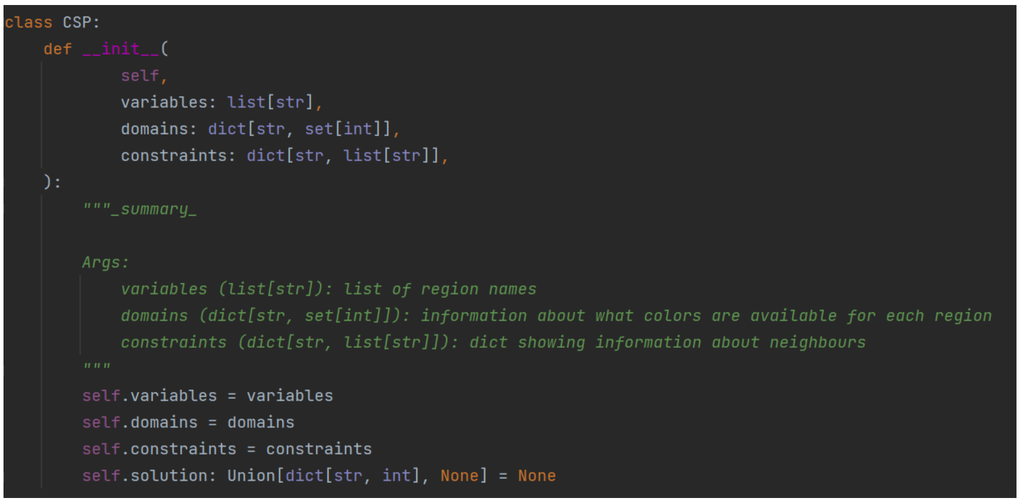
The backtracking algorithm uses heuristics to ensure that the solution is found if it exists, but the wrong choice of the primary regions can prolong the search of the algorithm and make it inefficient. Backtracking is a technique that will work properly only with small maps since the complexity of it is exponential. Forward checking reduces the search space by removing the wrong solutions early on, before checking them manually by the backtracking algorithm, which makes the convergence to the solution faster. Maintaining the search tree and intermediate results also consumes a significant amount of memory, which could require a lot of computational power for maps with a large number of regions.

2. Implementation

2.1. Implementation of the algorithm

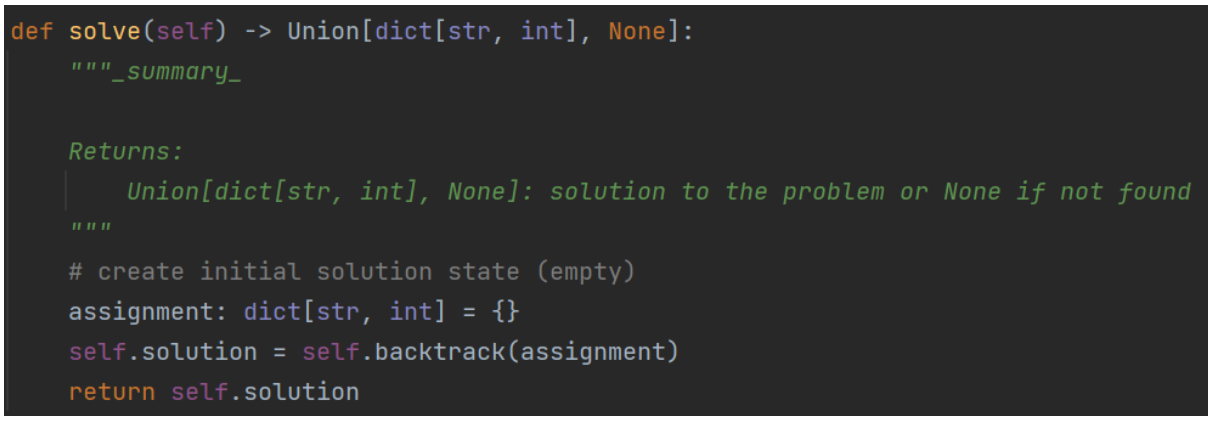
Firstly, we implement the CSP class, which represents the Constraint Satisfaction Problem. The snippets of code, with the methods included in this class, are depicted in Figures 2.1.1.-2.1.6. Method __*init*__ is the constructor method, used for the initialization of every parameter. It initializes attributes such as *variables*, *domains*, *constraints*, and *solutions*. The *variables* attribute is a list of map regions that must be colored. The *domains* attribute is a dictionary mapping each variable to a set of possible values (colors). The *constraints* attribute is a dictionary, in which the key is the region, and the values are lists of adjacent neighbors.

The next one, called *solve*, finds a solution to the Constraint Satisfaction Problem, using backtracking and forward checking. It initializes an empty dictionary called an *assignment,* which represents the current state of *variables*. Since the map is initially empty, the dictionary with the variables (regions) must also be empty. Then, the method calls the backtracking method, passing the assignment variable as an argument. The returned value of this method is assigned to the solution attribute of the CSP class.



```python
class CSP:
    def __init__(
            self,
            variables: list[str],
            domains: dict[str, set[int]],
            constraints: dict[str, list[str]],
    ):
        """_summary_

        Args:
            variables (list[str]): list of region names
            domains (dict[str, set[int]]): information about what colors are available for each region
            constraints (dict[str, list[str]]): dict showing information about neighbours
        """
        self.variables = variables
        self.domains = domains
        self.constraints = constraints
        self.solution: Union[dict[str, int], None] = None
```

Figure 2.1.1. - Code implementation of the method __*init*__ - initializing all attributes of the CSP class.



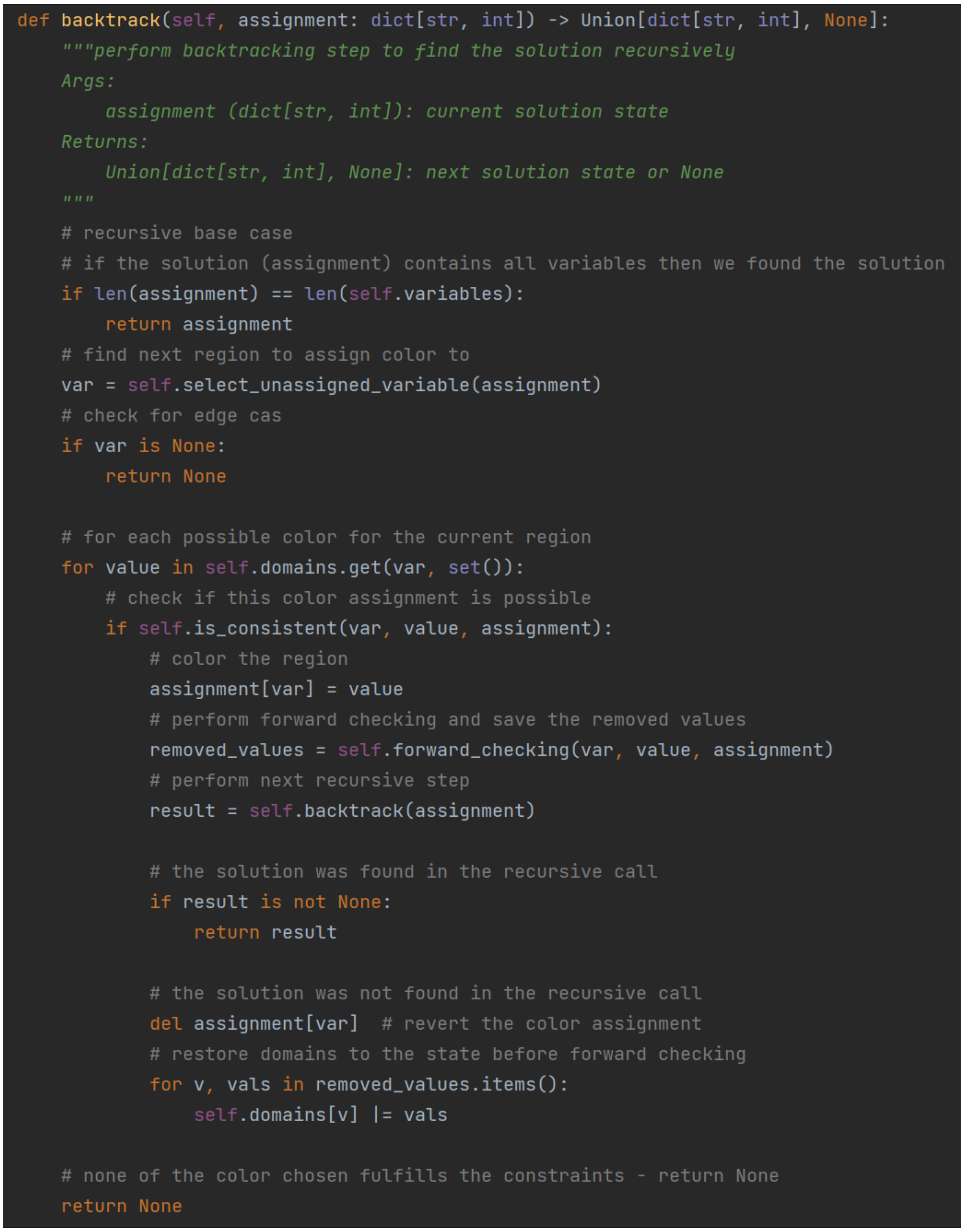
```python
def solve(self) -> Union[dict[str, int], None]:
    """_summary_

    Returns:
        Union[dict[str, int], None]: solution to the problem or None if not found
    """
    # create initial solution state (empty)
    assignment: dict[str, int] = {}
    self.solution = self.backtrack(assignment)
    return self.solution
```

Figure 2.1.2. - Code implementation of the method *solve*.

The most critical function, the *backtrack* method, is crucial for solving the CSP problem. The goal of the *backtrack* method is to find a valid assignment of colors to all regions that satisfies the constraint, which ensures that no two adjacent regions have the same color. If the method finds a valid solution to the CSP problem, it returns a dictionary representing the assignment of colors to regions. Each key-value pair in the dictionary represents a region and the color assigned to it. If it does not find a solution, it returns None. The backtrack method operates within the context of a CSP class instance, and it takes one parameter, which indicates the current state of the solution. The assignment dictionary provides a region as a key and a currently assigned color as the value. Every time, at the beginning of a recursive call, there is a check if the dictionary with the solution has every region with an assigned color. If this is the case, the algorithm returns the solution. Otherwise, it proceeds to find the next unassigned variable (region) of the map, calling the method *select_unassigned_variable*. This method takes the current solution as an argument and searches for the next region in the variables list. If it is not assigned already, it returns it. For each possible color (value) in the domain of the current variable, the algorithm checks if assigning that color to the variable is consistent with the constraints. It calls the *is_consistent* method to perform this check. If any other neighbor of the currently checked variable has the same color already assigned, the function returns False, otherwise, True. After the method *is_consistent,* returns True, it colors the region with the previously approved color. Now, the *forward_checking* method is being called.

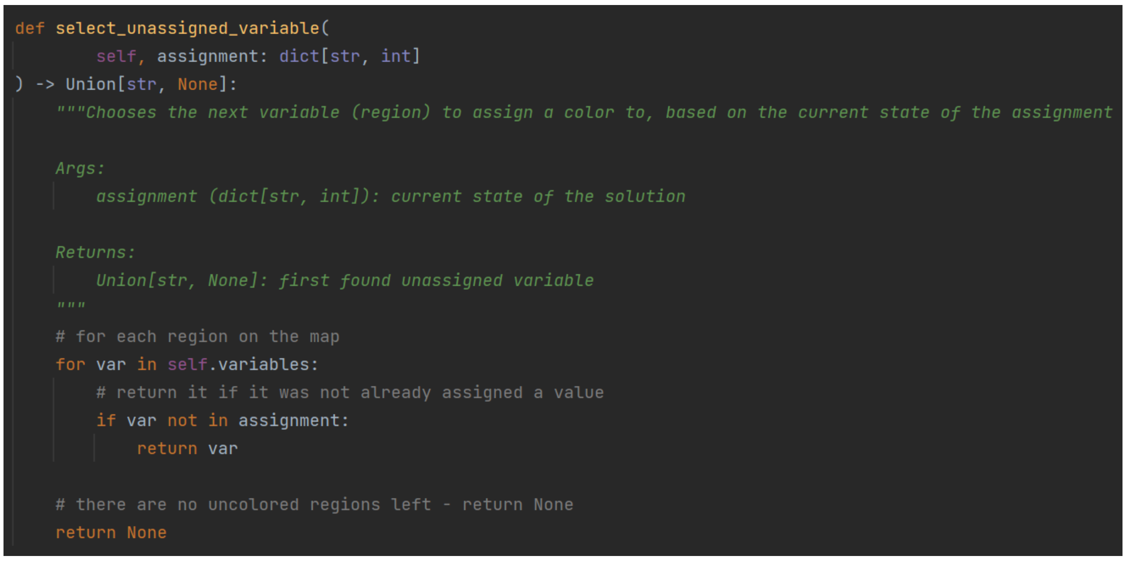
This method is called after a certain region has been assigned a color. It removes this color from all the neighboring region's domains, but saves the color in an additional dictionary, in case the algorithm has to backtrack and change the color of a current variable. The removed_values dictionary's keys are the consecutive regions, and the values are sets of removed colors.

After removing the colors that are no longer consistent with the map coloring problem rules, the algorithm gets back to the *backtrack* method and starts the next recursive call. The last few lines of the function are checking if the solution has been found. It checks if there is a possibility that every region has a neighbor with a different color assigned, given that the user inputs the number of colors that have to be checked during the execution of the program. The solution is output as a dictionary of regions as keys and their colors as values. If the solution has not been found, a current recursive call is being ended, which means the algorithm backtracks to the last step and checks if there is a result that is not *None*. Before every backtracking, the algorithm reverts the color assignment, by deleting the variable from the assignment dictionary, and also restores the deleted color domains to the neighbors of the current variable. The method can backtrack this way multiple times before finding a solution to the whole map or deciding that there is no possible solution.



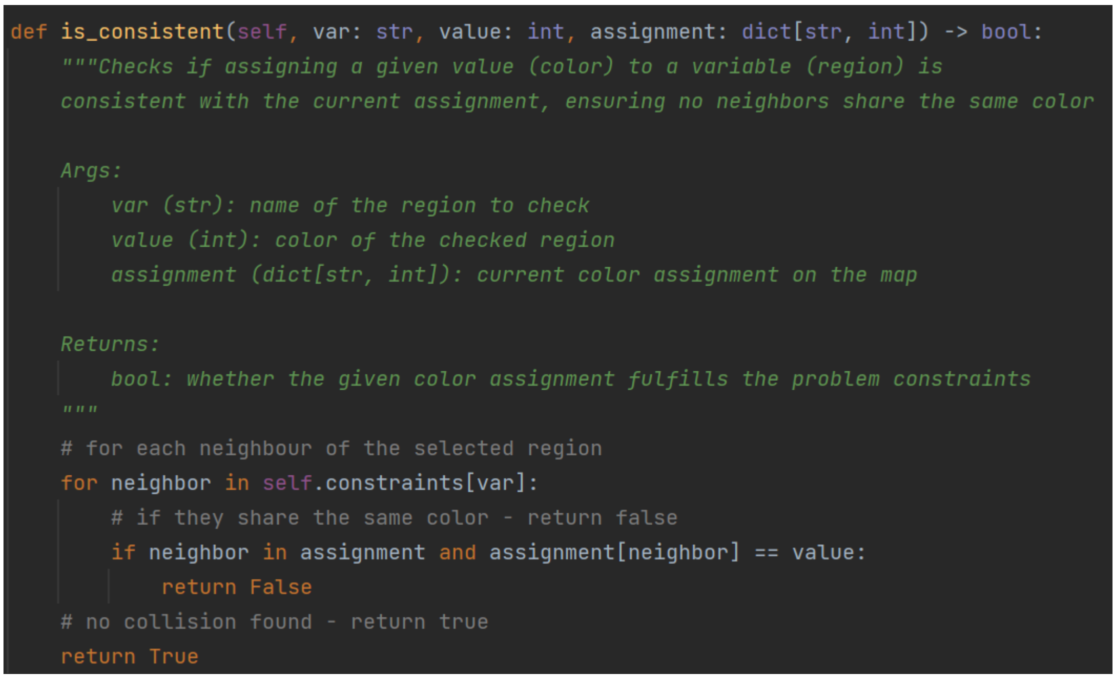
```python
def backtrack(self, assignment: dict[str, int]) -> Union[dict[str, int], None]:
    """perform backtracking step to find the solution recursively
    Args:
        assignment (dict[str, int]): current solution state
    Returns:
        Union[dict[str, int], None]: next solution state or None
    """
    # recursive base case
    # if the solution (assignment) contains all variables then we found the solution
    if len(assignment) == len(self.variables):
        return assignment
    # find next region to assign color to
    var = self.select_unassigned_variable(assignment)
    # check for edge cas
    if var is None:
        return None

    # for each possible color for the current region
    for value in self.domains.get(var, set()):
        # check if this color assignment is possible
        if self.is_consistent(var, value, assignment):
            # color the region
            assignment[var] = value
            # perform forward checking and save the removed values
            removed_values = self.forward_checking(var, value, assignment)
            # perform next recursive step
            result = self.backtrack(assignment)

            # the solution was found in the recursive call
            if result is not None:
                return result

            # the solution was not found in the recursive call
            del assignment[var]  # revert the color assignment
            # restore domains to the state before forward checking
            for v, vals in removed_values.items():
                self.domains[v] |= vals

    # none of the color chosen fulfills the constraints - return None
    return None
```

Figure 2.1.3. - Snippet of the code with the *backtrack* method.



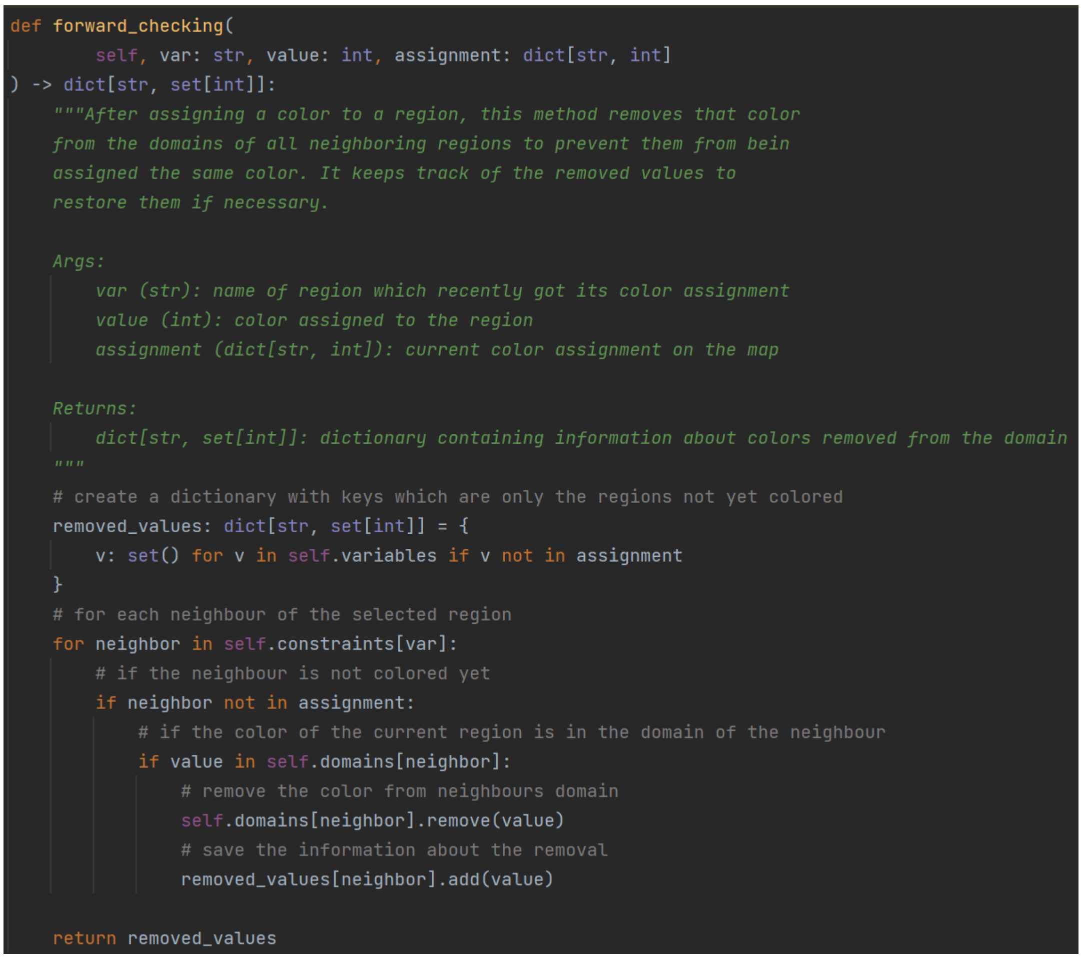
```python
def select_unassigned_variable(
        self, assignment: dict[str, int]
) -> Union[str, None]:
    """Chooses the next variable (region) to assign a color to, based on the current state of the assignment

    Args:
        assignment (dict[str, int]): current state of the solution

    Returns:
        Union[str, None]: first found unassigned variable
    """
    # for each region on the map
    for var in self.variables:
        # return it if it was not already assigned a value
        if var not in assignment:
            return var

    # there are no uncolored regions left - return None
    return None
```

Figure 2.1.4. - Snippet of the code with the *select_unassigned_variable* method.



```python
def is_consistent(self, var: str, value: int, assignment: dict[str, int]) -> bool:
    """Checks if assigning a given value (color) to a variable (region) is
    consistent with the current assignment, ensuring no neighbors share the same color

    Args:
        var (str): name of the region to check
        value (int): color of the checked region
        assignment (dict[str, int]): current color assignment on the map

    Returns:
        bool: whether the given color assignment fulfills the problem constraints
    """
    # for each neighbour of the selected region
    for neighbor in self.constraints[var]:
        # if they share the same color - return false
        if neighbor in assignment and assignment[neighbor] == value:
            return False
    # no collision found - return true
    return True
```

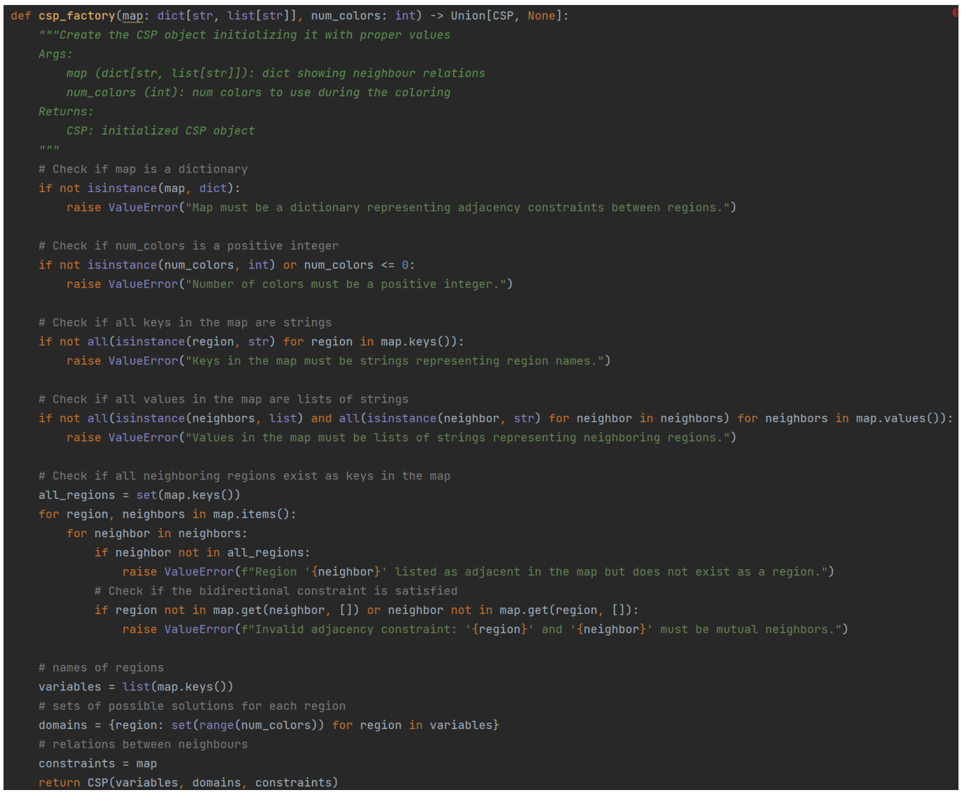
Figure 2.1.5. - Snippet of the code with the *is_consistent* method.



```python
def forward_checking(
        self, var: str, value: int, assignment: dict[str, int]
) -> dict[str, set[int]]:
    """After assigning a color to a region, this method removes that color
    from the domains of all neighboring regions to prevent them from bein
    assigned the same color. It keeps track of the removed values to
    restore them if necessary.

    Args:
        var (str): name of region which recently got its color assignment
        value (int): color assigned to the region
        assignment (dict[str, int]): current color assignment on the map

    Returns:
        dict[str, set[int]]: dictionary containing information about colors removed from the domain
    """
    # create a dictionary with keys which are only the regions not yet colored
    removed_values: dict[str, set[int]] = {
        v: set() for v in self.variables if v not in assignment
    }
    # for each neighbour of the selected region
    for neighbor in self.constraints[var]:
        # if the neighbour is not colored yet
        if neighbor not in assignment:
            # if the color of the current region is in the domain of the neighbour
            if value in self.domains[neighbor]:
                # remove the color from neighbours domain
                self.domains[neighbor].remove(value)
                # save the information about the removal
                removed_values[neighbor].add(value)

    return removed_values
```

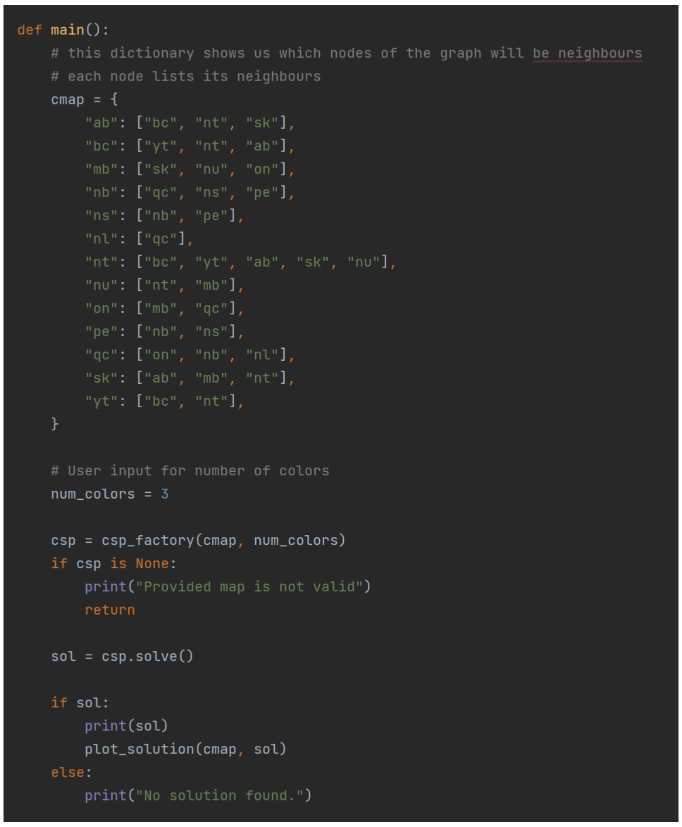
Figure 2.1.6. - Snippet of the code with the *forward_checking* method.

The program is creating the class CSP in the *main* function, by calling the function *csp_factory*. It not only creates an instance of the class but also checks the input for correctness. The user has to pass the dictionary, indicating all of the regions and the variables adjacent to them, as well as the number of colors that they want to create a solution for. The code of the csp_factory function is shown in Figure 2.1.7. This function firstly confirms if the map parameter is a dictionary, if the number of colors parameter is a positive integer, if all the keys in the map are strings, if all values in the map are lists of strings, if all neighboring regions are existing on their own in the map dictionary, and if both adjacent regions have each other in their neighbors sets. After that, the function returns a constructed object. The *main* function is shown in Figure 2.1.8. In this function, there is a declaration of all the regions and their neighbors in the form of a dictionary. The user also declares the number of colors that they want to experiment on. After constructing a CSP class object, the user checks if there is a solution for their parameters. Additionally, the program visualizes the problem in the form of a graph, in which every circle is a region, and every neighbor is another circle connected with an edge.



```python
def csp_factory(map: dict[str, list[str]], num_colors: int) -> Union[CSP, None]:
    """Create the CSP object initializing it with proper values
    Args:
        map (dict[str, list[str]]): dict showing neighbour relations
        num_colors (int): num colors to use during the coloring
    Returns:
        CSP: initialized CSP object
    """
    # Check if map is a dictionary
    if not isinstance(map, dict):
        raise ValueError("Map must be a dictionary representing adjacency constraints between regions.")

    # Check if num_colors is a positive integer
    if not isinstance(num_colors, int) or num_colors <= 0:
        raise ValueError("Number of colors must be a positive integer.")

    # Check if all keys in the map are strings
    if not all(isinstance(region, str) for region in map.keys()):
        raise ValueError("Keys in the map must be strings representing region names.")

    # Check if all values in the map are lists of strings
    if not all(isinstance(neighbors, list) and all(isinstance(neighbor, str) for neighbor in neighbors) for neighbors in map.values()):
        raise ValueError("Values in the map must be lists of strings representing neighboring regions.")

    # Check if all neighboring regions exist as keys in the map
    all_regions = set(map.keys())
    for region, neighbors in map.items():
        for neighbor in neighbors:
            if neighbor not in all_regions:
                raise ValueError(f"Region '{neighbor}' listed as adjacent in the map but does not exist as a region.")
            # Check if the bidirectional constraint is satisfied
            if region not in map.get(neighbor, []) or neighbor not in map.get(region, []):
                raise ValueError(f"Invalid adjacency constraint: '{region}' and '{neighbor}' must be mutual neighbors.")

    # names of regions
    variables = list(map.keys())
    # sets of possible solutions for each region
    domains = {region: set(range(num_colors)) for region in variables}
    # relations between neighbours
    constraints = map
    return CSP(variables, domains, constraints)
```

Figure 2.1.7. - Snippet of the code with the *csp_factory* function.



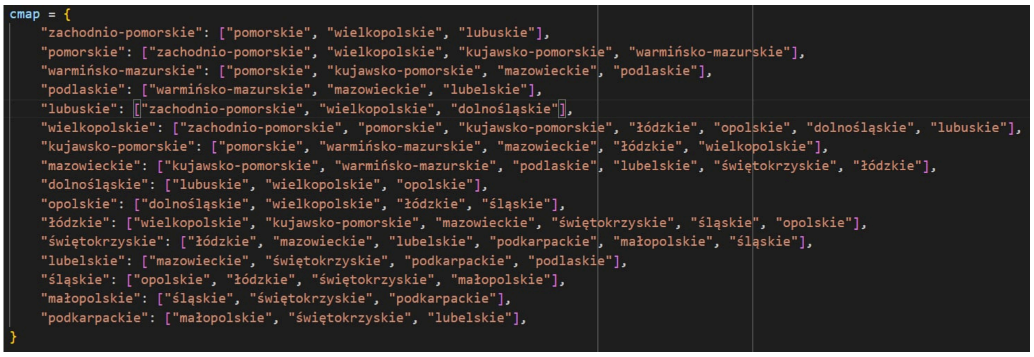
```python
def main():
    # this dictionary shows us which nodes of the graph will be neighbours
    # each node lists its neighbours
    cmap = {
        "ab": ["bc", "nt", "sk"],
        "bc": ["yt", "nt", "ab"],
        "mb": ["sk", "nu", "on"],
        "nb": ["qc", "ns", "pe"],
        "ns": ["nb", "pe"],
        "nl": ["qc"],
        "nt": ["bc", "yt", "ab", "sk", "nu"],
        "nu": ["nt", "mb"],
        "on": ["mb", "qc"],
        "pe": ["nb", "ns"],
        "qc": ["on", "nb", "nl"],
        "sk": ["ab", "mb", "nt"],
        "yt": ["bc", "nt"],
    }

    # User input for number of colors
    num_colors = 3

    csp = csp_factory(cmap, num_colors)
    if csp is None:
        print("Provided map is not valid")
        return

    sol = csp.solve()

    if sol:
        print(sol)
        plot_solution(cmap, sol)
    else:
        print("No solution found.")
```

Figure 2.1.8. - Snippet of the code with the *main* function.

3. Discussion and results

3.1. Testing General Correctness of the Implementation

To test the general correctness of the implementation, we have created two test cases, each with a number of regions greater than 10. The program outputs the solution as a list of regions, and for each variable, it assigns a number that indicates the color that the region has to be filled with. The code also provides a visualization of the regions depicted in a graph form. Based on this, we drew two maps of the exemplary solutions. These maps are depicted in Figures 3.1.2. and 3.1.6. The graphs of these examples are shown in Figures 3.1.3. and 3.1.6. The respective input dictionaries with the regions and their neighbors are provided in Figures 3.1.1. and 3.1.5. The console outputs of the code of these two examples are shown in Figures 3.1.4 and 3.1.8.



```
cmap = {
    "zachodnio-pomorskie": ["pomorskie", "wielkopolskie", "lubuskie"],
    "pomorskie": ["zachodnio-pomorskie", "wielkopolskie", "kujawsko-pomorskie", "warmińsko-mazurskie"],
    "warmińsko-mazurskie": ["pomorskie", "kujawsko-pomorskie", "mazowieckie", "podlaskie"],
    "podlaskie": ["warmińsko-mazurskie", "mazowieckie", "lubelskie"],
    "lubuskie": ["zachodnio-pomorskie", "wielkopolskie", "dolnośląskie"],
    "wielkopolskie": ["zachodnio-pomorskie", "pomorskie", "kujawsko-pomorskie", "łódzkie", "opolskie", "dolnośląskie", "lubuskie"],
    "kujawsko-pomorskie": ["pomorskie", "warmińsko-mazurskie", "mazowieckie", "łódzkie", "wielkopolskie"],
    "mazowieckie": ["kujawsko-pomorskie", "warmińsko-mazurskie", "podlaskie", "lubelskie", "świętokrzyskie", "łódzkie"],
    "dolnośląskie": ["lubuskie", "wielkopolskie", "opolskie"],
    "opolskie": ["dolnośląskie", "wielkopolskie", "łódzkie", "śląskie"],
    "łódzkie": ["wielkopolskie", "kujawsko-pomorskie", "mazowieckie", "świętokrzyskie", "śląskie", "opolskie"],
    "świętokrzyskie": ["łódzkie", "mazowieckie", "lubelskie", "podkarpackie", "małopolskie", "śląskie"],
    "lubelskie": ["mazowieckie", "świętokrzyskie", "podkarpackie", "podlaskie"],
    "śląskie": ["opolskie", "łódzkie", "świętokrzyskie", "małopolskie"],
    "małopolskie": ["śląskie", "świętokrzyskie", "podkarpackie"],
    "podkarpackie": ["małopolskie", "świętokrzyskie", "lubelskie"],
}
```

Figure 3.1.1. - The map input of the first example.

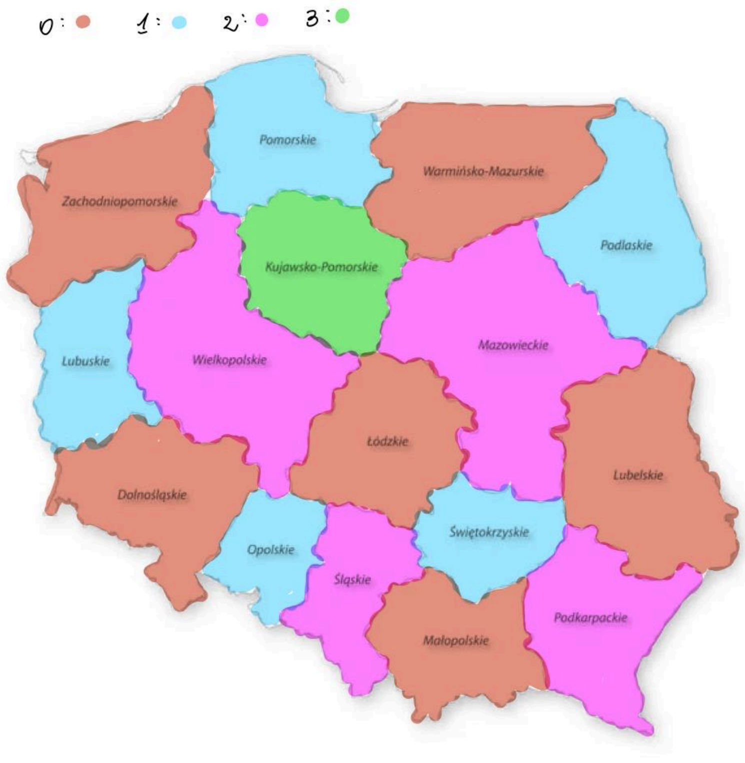


Figure 3.1.2. - The map drawing of the first example.

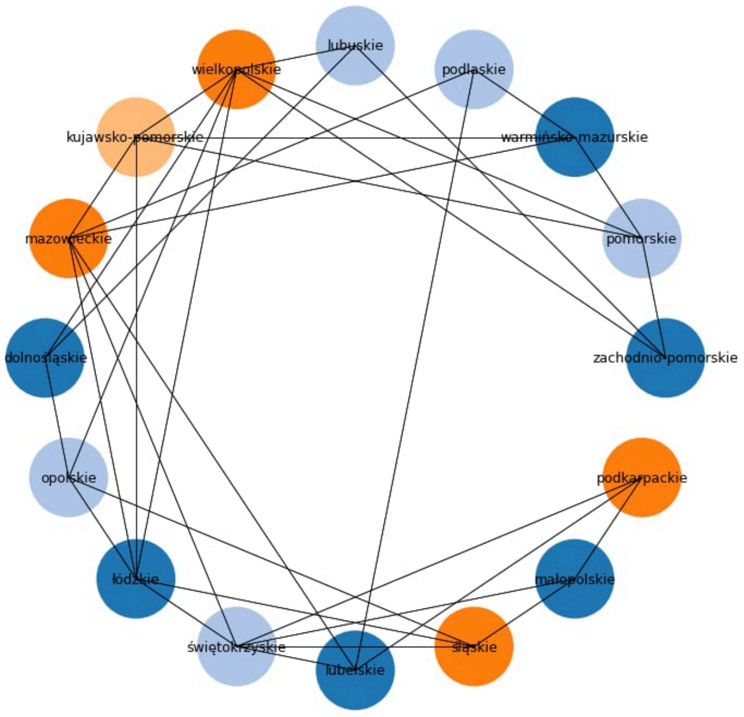
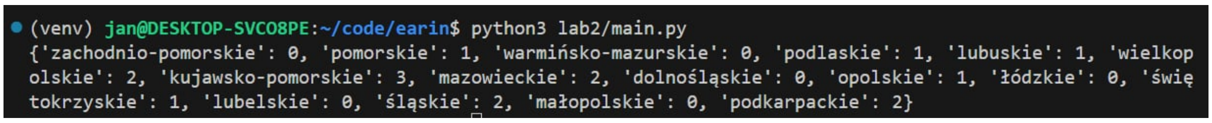
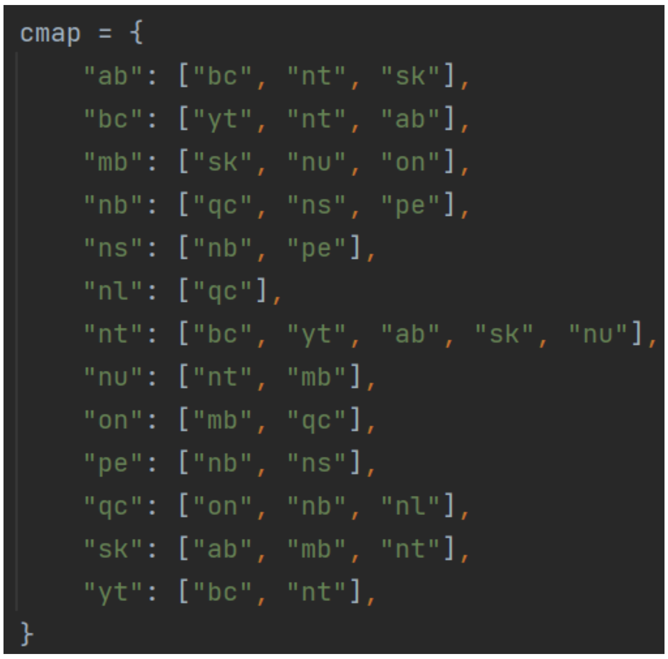


Figure 3.1.3. - The graph visualization of the first example.



```
● (venv) jan@DESKTOP-SVCO8PE:~/code/earin$ python3 lab2/main.py
  {'zachodnio-pomorskie': 0, 'pomorskie': 1, 'warmińsko-mazurskie': 0, 'podlaskie': 1, 'lubuskie': 1, 'wielkop
  olskie': 2, 'kujawsko-pomorskie': 3, 'mazowieckie': 2, 'dolnośląskie': 0, 'opolskie': 1, 'łódzkie': 0, 'świę
  tokrzyskie': 1, 'lubelskie': 0, 'śląskie': 2, 'małopolskie': 0, 'podkarpackie': 2}
```

Figure 3.1.4. - The console output of the first example.



```
cmap = {
    "ab": ["bc", "nt", "sk"],
    "bc": ["yt", "nt", "ab"],
    "mb": ["sk", "nu", "on"],
    "nb": ["qc", "ns", "pe"],
    "ns": ["nb", "pe"],
    "nl": ["qc"],
    "nt": ["bc", "yt", "ab", "sk", "nu"],
    "nu": ["nt", "mb"],
    "on": ["mb", "qc"],
    "pe": ["nb", "ns"],
    "qc": ["on", "nb", "nl"],
    "sk": ["ab", "mb", "nt"],
    "yt": ["bc", "nt"],
}
```

Figure 3.1.5. - The map input of the second example.

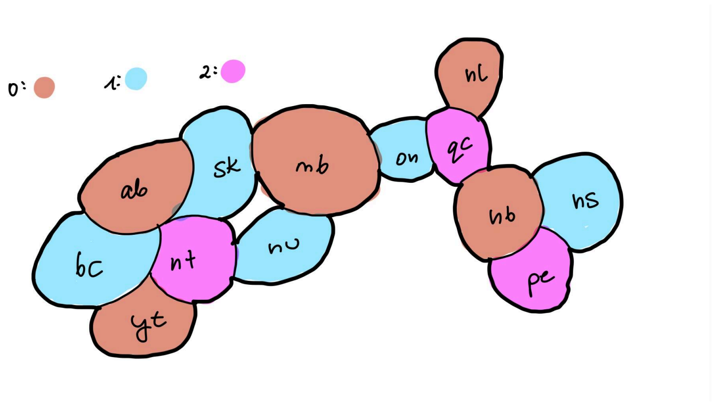


Figure 3.1.6. - The map drawing of the second example.

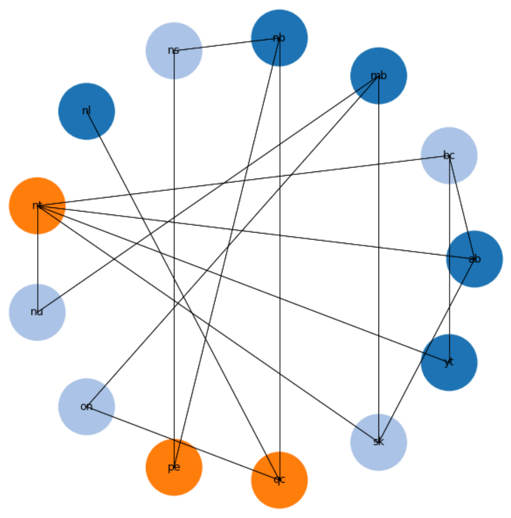
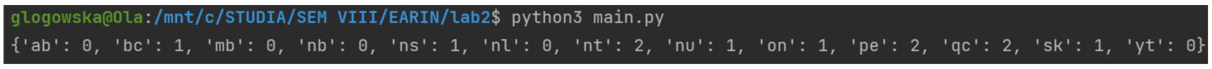


Figure 3.1.7. - The graph visualization of the second example.



```
glogowska@Ola:/mnt/c/STUDIA/SEM VIII/EARIN/lab2$ python3 main.py
{'ab': 0, 'bc': 1, 'mb': 0, 'nb': 0, 'ns': 1, 'nl': 0, 'nt': 2, 'nu': 1, 'on': 1, 'pe': 2, 'qc': 2, 'sk': 1, 'yt': 0}
```

Figure 3.1.8. - The console output of the second example.

3.2. Further Testing of Correctness on Corner Cases

Disjoint graphs

The program should behave correctly when a map containing disjoint regions is provided. Such cases could occur, for example while coloring a map of a country located on several islands, such as Japan.

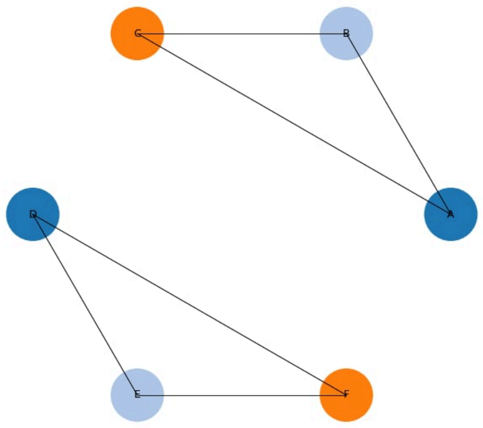
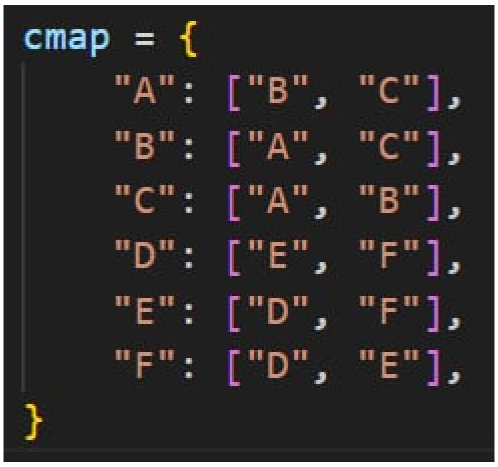
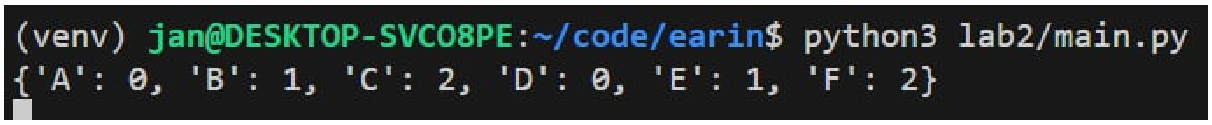


Figure 3.2.1. - Graph showing the coloring solution for disjoint regions



```
cmap = {
    "A": ["B", "C"],
    "B": ["A", "C"],
    "C": ["A", "B"],
    "D": ["E", "F"],
    "E": ["D", "F"],
    "F": ["D", "E"],
}
```

Figure 3.2.2. - Input for the disjoint regions edge case

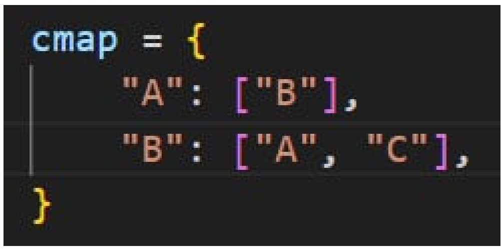


```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin$ python3 lab2/main.py
{'A': 0, 'B': 1, 'C': 2, 'D': 0, 'E': 1, 'F': 2}
```

Figure 3.2.3 - The console output for disjoint regions edge case.

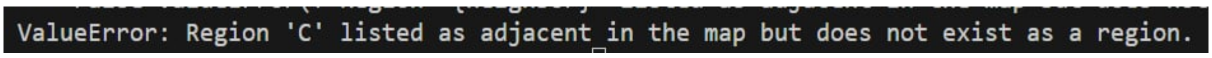
Nonexistent neighbor

When specifying the relations, all of the listed neighbors should appear in the data structure. If they do not exist, generating the solution is impossible.



```
cmap = {
    "A": ["B"],
    "B": ["A", "C"],
}
```

Figure 3.2.4. - Input for the nonexistent neighbor edge case.

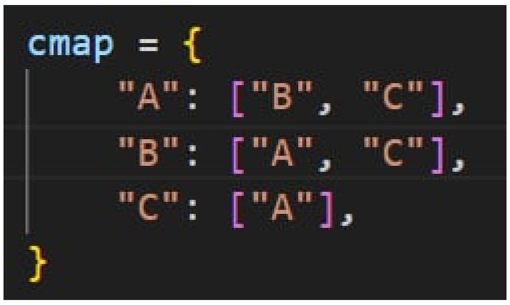


```
ValueError: Region 'C' listed as adjacent in the map but does not exist as a region.
```

Figure 3.2.5. - The console output for the non existent neighbor edge case.

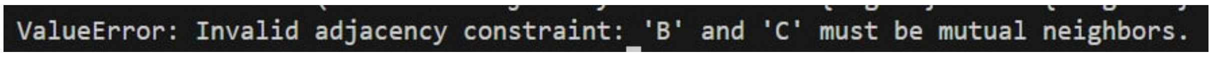
The relationship is not bidirectional

When we initialize the data structure, all of the neighbor relationships should be defined both ways. In other words, if A is adjacent to B, then B is adjacent to A. Otherwise, the data structure is not valid.



```
cmap = {
    "A": ["B", "C"],
    "B": ["A", "C"],
    "C": ["A"],
}
```

Figure 3.2.6. - Input for the invalid bidirectional relationship edge case.

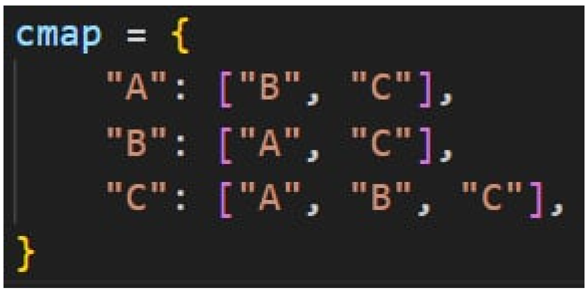


```
ValueError: Invalid adjacency constraint: 'B' and 'C' must be mutual neighbors.
```

Figure 3.2.6. - The console output for the invalid bidirectional relationship edge case.

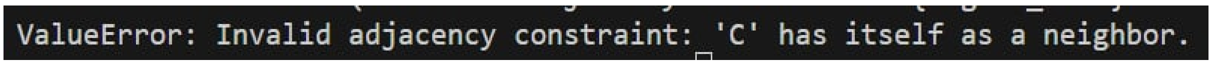
Element adjacent to itself

A region on the map cannot be adjacent to itself, so if the given data structure contains such a case, then it is not valid.



```
cmap = {
    "A": ["B", "C"],
    "B": ["A", "C"],
    "C": ["A", "B", "C"],
}
```

Figure 3.2.7. - Input for element adjacent to itself edge case.



```
ValueError: Invalid adjacency constraint: 'C' has itself as a neighbor.
```

Figure 3.2.8. - The console output for the element adjacent to itself edge case.

4. Conclusions

The implementation was tested with maps having more than 10 regions, demonstrating the algorithm's capability to find valid coloring solutions efficiently and proving the general correctness of the program. However, additionally, the handling of corner cases, like disjoint graphs, nonexistent neighbors, and non-bidirectional relationships between regions, were also extensively tested.

By eliminating values inconsistent with the problem constraints from the domains of adjacent regions, forward checking can often significantly reduce the search space, leading to finding the solution in fewer computational steps. It helps in the early detection of dead ends and avoids unnecessary exploration of doomed paths, saving a lot of computational resources. However, by updating the domains of all unassigned variables that are neighbors of the newly assigned region, we trade time for memory every time a variable is assigned. We are able to find the solution faster, but our program uses more resources. For small graphs where the nodes do not have a lot of neighbors, it might not be a problem. However, in cases where the graph is very large, and the nodes are densely connected, we might run out of available program memory.