# TASK 1:

def is\_valid\_assignment(region, color, coloring, constraints):

    # Check for neighbors color

    for neighbor in constraints[region]:

        if color == coloring.get(neighbor):

            return False  # A neighbor has the same color, so the assignment is not valid.

    return True  # No neighbors have the same color, so the assignment is valid.

def assign\_colors(constraints, colors, regions, coloring={}, current\_region\_index=0):

    # assigining colours

    if current\_region\_index == len(regions):

        return coloring  # Successfully assigned colors to all regions.

    region = regions[current\_region\_index]  # Current region to assign a color to.

    for color in colors:

        if is\_valid\_assignment(region, color, coloring, constraints):

            coloring[region] = color  # Temporarily assign the color to the region.

            if assign\_colors(constraints, colors, regions, coloring, current\_region\_index + 1):

                return coloring  # Successfully found a valid coloring for the entire map.

            # If not successful, backtrack by removing the color assignment.

            del coloring[region]

    # If no valid color assignment is found for the current region, backtrack to the previous region.

    return None

# Define the constraints based on region adjacency.

constraints = {

    "Pb": ["Sd", "KP", "Bl", "GB", "AJK"],

    "Sd": ["Pb", "Bl"],

    "KP": ["Pb", "Bl", "AJK"],  # Assuming KP is adjacent to Bl and AJK for completeness.

    "Bl": ["Pb", "Sd", "KP"],

    "GB": ["Pb", "KP", "AJK"],  # Assuming GB is adjacent to KP for completeness.

    "AJK": ["Pb", "KP", "AJK"],  # Assuming AJK is adjacent to KP for completeness.

}

# Define the list of regions.

regions = ["Pb", "Sd", "KP", "Bl", "GB", "AJK"]

# Define the available colors.

colors = ["Yellow", "Orange", "Purple"]

# Attempt to find a valid coloring for the map.

coloring\_solution = assign\_colors(constraints, colors, regions)

if coloring\_solution:

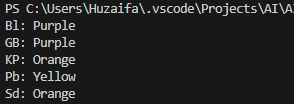
    for region, color in sorted(coloring\_solution.items()):

        print(f"{region}: {color}")

else:

    print("No valid coloring solution found.")

# OUTPUT:



# TASK 2:

def AC3(csp):

    queue = [(Xi, Xj) for Xi in csp['variables'] for Xj in csp['neighbors'][Xi]]

    while queue:

        (Xi, Xj) = queue.pop(0)

        if revise(csp, Xi, Xj):

            if len(csp['domains'][Xi]) == 0:

                return False  # Failure due to no legal values

            for Xk in csp['neighbors'][Xi]:

                if Xk != Xj:

                    queue.append((Xk, Xi))

    return True  # CSP is arc-consistent

def revise(csp, Xi, Xj):

    revised = False

    for x in csp['domains'][Xi][:]:

        # If no value y in Dj allows (x, y) to satisfy the constraint between Xi and Xj,

        # then delete x from Di.

        if not any((x != y) for y in csp['domains'][Xj]):

            csp['domains'][Xi].remove(x)

            revised = True

    return revised

def select\_unassigned\_variable(csp, assignment):

    # Implementing MRV heuristic

    unassigned\_variables = [var for var in csp['variables'] if var not in assignment]

    return min(unassigned\_variables, key=lambda var: len(csp['domains'][var]))

def order\_domain\_values(var, assignment, csp):

    # Implementing LCV heuristic

    return sorted(csp['domains'][var], key=lambda val: num\_conflicts(var, val, csp, assignment))

def num\_conflicts(var, val, csp, assignment):

    # Counts conflicts to implement LCV

    count = 0

    for neighbor in csp['neighbors'][var]:

        if neighbor in assignment and assignment[neighbor] == val:

            count += 1

    return count

def is\_consistent(var, value, assignment, csp):

    for neighbor in csp['neighbors'][var]:

        if neighbor in assignment and assignment[neighbor] == value:

            return False

    return True

def forward\_checking(csp, var, value, assignment):

    for neighbor in csp['neighbors'][var]:

        if neighbor not in assignment:

            csp['domains'][neighbor] = [x for x in csp['domains'][neighbor] if x != value]

            if len(csp['domains'][neighbor]) == 0:

                return False  # Failure due to wipeout

    return True

def backtrack(assignment, csp):

    if len(assignment) == len(csp['variables']):

        return assignment  # Success

    var = select\_unassigned\_variable(csp, assignment)

    for value in order\_domain\_values(var, assignment, csp):

        if is\_consistent(var, value, assignment, csp):

            assignment[var] = value

            # Apply forward checking

            if forward\_checking(csp, var, value, assignment):

                result = backtrack(assignment.copy(), csp)

                if result:

                    return result

            del assignment[var]

    return False  # Failure

def backtracking\_search(csp):

    return backtrack({}, csp)