```python

from .util import (count, first)

from .problem import Problem

class CSP(Problem):
    """This class describes finite-domain Constraint Satisfaction Problems.
    A CSP is specified by the following inputs:
        variables   A list of variables; each is atomic (e.g. int or string).
        domains     A dict of {var:[possible_value, ...]} entries.
        neighbors   A dict of {var:[var,...]} that for each variable lists
                    the other variables that participate in constraints.
        constraints A function f(A, a, B, b) that returns true if neighbors
                    A, B satisfy the constraint when they have values A=a, B=b

    In the textbook and in most mathematical definitions, the
    constraints are specified as explicit pairs of allowable values,
    but the formulation here is easier to express and more compact for
    most cases. (For example, the n-Queens problem can be represented
    in O(n) space using this notation, instead of O(N^4) for the
    explicit representation.) In terms of describing the CSP as a
    problem, that's all there is.

    However, the class also supports data structures and methods that help you
    solve CSPs by calling a search function on the CSP. Methods and slots are
    as follows, where the argument 'a' represents an assignment, which is a
    dict of {var:val} entries:
        assign(var, val, a)     Assign a[var] = val; do other bookkeeping
        unassign(var, a)        Do del a[var], plus other bookkeeping
        nconflicts(var, val, a) Return the number of other variables that
                                conflict with var=val
        curr_domains[var]       Slot: remaining consistent values for var
                                Used by constraint propagation routines.
    The following methods are used only by graph_search and tree_search:
        actions(state)          Return a list of actions
        result(state, action)   Return a successor of state
        goal_test(state)        Return true if all constraints satisfied
    The following are just for debugging purposes:
        nassigns                Slot: tracks the number of assignments made
        display(a)              Print a human-readable representation

    The following methods are for supporting any type of domain restriction
    (pruning of domains), such as is done in constraint propagation:

    support_pruning() - Initializes the domains of all variables
        MUST BE CALLED before starting to prune, is called automatically
        the first time suppose is called
    suppose(var, value) - Suppose that variable var = value.  Returns a list
        of values removed [(var, val1), (var, val2), ...]
    prune(var, value, removed_list) - Rule out value for specified variable
        If removed_list is not None, (var, value) is appended to the list
    choices(var) - List values remaining in domain
    infer_assignment() - Assign variables whose domain has been reduced
        to a single value
    restore(removals) - Given a list of pruned values [(var, val), ...],
        restore these values to their variable's domain
    conflicted_vars(current) - Given a current set of assignments, return
        the set of variables that are in conflict.
    """

    def __init__(self, variables, domains, neighbors, constraints):
        """Construct a CSP problem. If variables is empty, it becomes domains.keys()."""
        variables = variables or list(domains.keys())

        self.variables = variables
        self.domains = domains
        self.neighbors = neighbors
        self.constraints = constraints
        self.initial = ()
        self.curr_domains = None
        self.nassigns =

    def assign(self, var, val, assignment):
        """Add {var: val} to assignment; Discard the old value if any."""
        assignment[var] = val
        self.nassigns +=

```

```
    def unassign(self, var, assignment):
        """Remove {var: val} from assignment.
        DO NOT call this if you are changing a variable to a new value;
        just call assign for that."""
        if var in assignment:
            del assignment[var]

    def nconflicts(self, var, val, assignment):
        """Return the number of conflicts var=val has with other variables."""
        # Subclasses may implement this more efficiently
        def conflict(var2):
            return (var2 in assignment and
                    not self.constraints(var, val, var2, assignment[var2]))
        return count(conflict(v) for v in self.neighbors[var])

    def display(self, assignment):
        """Show a human-readable representation of the CSP."""
        # Subclasses can print in a prettier way, or display with a GUI
        print('CSP:', self, 'with assignment:', assignment)

    # These methods are for the tree and graph-search interface:

    def actions(self, state):
        """Return a list of applicable actions: nonconflicting
        assignments to an unassigned variable."""
        if len(state) == len(self.variables):
            return []
        else:
            assignment = dict(state)
            var = first([v for v in self.variables if v not in assignment])
            return [(var, val) for val in self.domains[var]
                    if self.nconflicts(var, val, assignment) ==  ]

    def result(self, state, action):
        """Perform an action and return the new state."""
        (var, val) = action
        return state + ((var, val),)

    def goal_test(self, state):
        """The goal is to assign all variables, with all constraints satisfied."""
        assignment = dict(state)
        return (len(assignment) == len(self.variables)
                and all(self.nconflicts(variables, assignment[variables], assignment) ==
                        for variables in self.variables))

    # These are for constraint propagation

    def support_pruning(self):
        """Make sure we can prune values from domains. (We want to pay
        for this only if we use it.)"""
        if self.curr_domains is None:
            self.curr_domains = {v: list(self.domains[v]) for v in self.variables}

    def suppose(self, var, value):
        """Start accumulating inferences from assuming var=value."""
        self.support_pruning()
        removals = [(var, a) for a in self.curr_domains[var] if a != value]
        self.curr_domains[var] = [value]
        return removals

    def prune(self, var, value, removals):
        """Rule out var=value."""
        self.curr_domains[var].remove(value)
        if removals is not None:
            removals.append((var, value))

    def choices(self, var):
        """Return all values for var that aren't currently ruled out."""
        return (self.curr_domains or self.domains)[var]

    def infer_assignment(self):
        """Return the partial assignment implied by the current inferences."""
        self.support_pruning()
        return {v: self.curr_domains[v][ ]
                for v in self.variables if   == len(self.curr_domains[v])}

    def restore(self, removals):
```

```python
        """Undo a supposition and all inferences from it."""
        for B, b in removals:
            self.curr_domains[B].append(b)

    # This is for min_conflicts search

    def conflicted_vars(self, current):
        """Return a list of variables in current assignment that are in conflict"""
        return [var for var in self.variables
                if self.nconflicts(var, current[var], current) >   ]

```