```
import collections, util, copy

############################################################
# Problem 0

# Hint: Take a look at the CSP class and the CSP examples in util.py
def create_chain_csp(n):
    # same domain for each variable
    domain = [0, 1]
    # name variables as x_1, x_2, ..., x_n
    variables = ['x%d'%i for i in range(1, n + 1)]
    csp = util.CSP()
    # Problem 0c
    # BEGIN_YOUR_CODE (our solution is 4 lines of code, but don't worry if you
deviate from this)
    for var in variables:
        csp.add_variable(var, domain)
    for i in range(n - 1):
        csp.add_binary_factor(variables[i], variables[i + 1], lambda x, y: x !=
y)
    # END_YOUR_CODE
    return csp


############################################################
# Problem 1

def create_nqueens_csp(n = 8):
    """
    Return an N-Queen problem on the board of size |n| * |n|.
    You should call csp.add_variable() and csp.add_binary_factor().

    @param n: number of queens, or the size of one dimension of the board.

    @return csp: A CSP problem with correctly configured factor tables
        such that it can be solved by a weighted CSP solver.
    """
    csp = util.CSP()
    # Problem 1a
    # BEGIN_YOUR_CODE (our solution is 7 lines of code, but don't worry if you
deviate from this)
    variables = ['x%d'%i for i in range(1, n + 1)]
    for index, var in enumerate(variables):
        csp.add_variable(var, [(index, i) for i in range(0, n)])
    for index1, var1 in enumerate(variables):
        for index2, var2 in enumerate(variables):
            if index1 != index2:
                csp.add_binary_factor(var1, var2, lambda x, y : x[1] != y[1] and
x[0] != y[0])
                csp.add_binary_factor(var1, var2, lambda x, y : (x[0] - x[1]) !=
(y[0] - y[1]))
                csp.add_binary_factor(var1, var2, lambda x, y : (x[0] + x[1]) !=
(y[0] + y[1]))
    # END_YOUR_CODE
    return csp

# A backtracking algorithm that solves weighted CSP.
# Usage:
#   search = BacktrackingSearch()
#   search.solve(csp)
class BacktrackingSearch():

    def reset_results(self):
        """
```

```
        This function resets the statistics of the different aspects of the
        CSP solver. We will be using the values here for grading, so please
        do not make any modification to these variables.
        """
        # Keep track of the best assignment and weight found.
        self.optimalAssignment = {}
        self.optimalWeight = 0

        # Keep track of the number of optimal assignments and assignments. These
        # two values should be identical when the CSP is unweighted or only has
binary
        # weights.
        self.numOptimalAssignments = 0
        self.numAssignments = 0

        # Keep track of the number of times backtrack() gets called.
        self.numOperations = 0

        # Keep track of the number of operations to get to the very first
successful
        # assignment (doesn't have to be optimal).
        self.firstAssignmentNumOperations = 0

        # List of all solutions found.
        self.allAssignments = []

    def print_stats(self):
        """
        Prints a message summarizing the outcome of the solver.
        """
        if self.optimalAssignment:
            print(("Found %d optimal assignments with weight %f in %d
operations" % \
                (self.numOptimalAssignments, self.optimalWeight,
self.numOperations)))
            print(("First assignment took %d operations" %
self.firstAssignmentNumOperations))
        else:
            print("No solution was found.")

    def get_delta_weight(self, assignment, var, val):
        """
        Given a CSP, a partial assignment, and a proposed new value for a
variable,
        return the change of weights after assigning the variable with the
proposed
        value.

        @param assignment: A dictionary of current assignment. Unassigned
variables
            do not have entries, while an assigned variable has the assigned
value
            as value in dictionary. e.g. if the domain of the variable A is
[5,6],
            and 6 was assigned to it, then assignment[A] == 6.
        @param var: name of an unassigned variable.
        @param val: the proposed value.

        @return w: Change in weights as a result of the proposed assignment.
This
            will be used as a multiplier on the current weight.
        """
        assert var not in assignment
        w = 1.0
```

```
        if self.csp.unaryFactors[var]:
            w *= self.csp.unaryFactors[var][val]
            if w == 0: return w
        for var2, factor in list(self.csp.binaryFactors[var].items()):
            if var2 not in assignment: continue  # Not assigned yet
            w *= factor[val][assignment[var2]]
            if w == 0: return w
        return w

    def solve(self, csp, mcv = False, ac3 = False):
        """
        Solves the given weighted CSP using heuristics as specified in the
        parameter. Note that unlike a typical unweighted CSP where the search
        terminates when one solution is found, we want this function to find
        all possible assignments. The results are stored in the variables
        described in reset_result().

        @param csp: A weighted CSP.
        @param mcv: When enabled, Most Constrained Variable heuristics is used.
        @param ac3: When enabled, AC-3 will be used after each assignment of an
            variable is made.
        """
        # CSP to be solved.
        self.csp = csp

        # Set the search heuristics requested asked.
        self.mcv = mcv
        self.ac3 = ac3

        # Reset solutions from previous search.
        self.reset_results()

        # The dictionary of domains of every variable in the CSP.
        self.domains = {var: list(self.csp.values[var]) for var in
self.csp.variables}

        # Perform backtracking search.
        self.backtrack({}, 0, 1)
        # Print summary of solutions.
        self.print_stats()

    def backtrack(self, assignment, numAssigned, weight):
        """
        Perform the back-tracking algorithms to find all possible solutions to
        the CSP.

        @param assignment: A dictionary of current assignment. Unassigned
variables
            do not have entries, while an assigned variable has the assigned
value
            as value in dictionary. e.g. if the domain of the variable A is
[5,6],
            and 6 was assigned to it, then assignment[A] == 6.
        @param numAssigned: Number of currently assigned variables
        @param weight: The weight of the current partial assignment.
        """

        self.numOperations += 1
        assert weight > 0
        if numAssigned == self.csp.numVars:
            # A satisfiable solution have been found. Update the statistics.
            self.numAssignments += 1
            newAssignment = {}
            for var in self.csp.variables:
```

```
                newAssignment[var] = assignment[var]
            self.allAssignments.append(newAssignment)

            if len(self.optimalAssignment) == 0 or weight >= self.optimalWeight:
                if weight == self.optimalWeight:
                    self.numOptimalAssignments += 1
                else:
                    self.numOptimalAssignments = 1
                self.optimalWeight = weight

                self.optimalAssignment = newAssignment
                if self.firstAssignmentNumOperations == 0:
                    self.firstAssignmentNumOperations = self.numOperations
            return

        # Select the next variable to be assigned.
        var = self.get_unassigned_variable(assignment)
        # Get an ordering of the values.
        ordered_values = self.domains[var]

        # Continue the backtracking recursion using |var| and |ordered_values|.
        if not self.ac3:
            # When arc consistency check is not enabled.
            for val in ordered_values:
                deltaWeight = self.get_delta_weight(assignment, var, val)
                if deltaWeight > 0:
                    assignment[var] = val
                    self.backtrack(assignment, numAssigned + 1, weight *
deltaWeight)
                    del assignment[var]
        else:
            # Arc consistency check is enabled.
            # Problem 1c: skeleton code for AC-3
            # You need to implement arc_consistency_check().
            for val in ordered_values:
                deltaWeight = self.get_delta_weight(assignment, var, val)
                if deltaWeight > 0:
                    assignment[var] = val
                    # create a deep copy of domains as we are going to look
                    # ahead and change domain values
                    localCopy = copy.deepcopy(self.domains)
                    # fix value for the selected variable so that hopefully we
                    # can eliminate values for other variables
                    self.domains[var] = [val]

                    # enforce arc consistency
                    self.arc_consistency_check(var)

                    self.backtrack(assignment, numAssigned + 1, weight *
deltaWeight)
                    # restore the previous domains
                    self.domains = localCopy
                    del assignment[var]

    def get_unassigned_variable(self, assignment):
        """
        Given a partial assignment, return a currently unassigned variable.

        @param assignment: A dictionary of current assignment. This is the same
as
            what you've seen so far.

        @return var: a currently unassigned variable.
        """
```

```
        if not self.mcv:
            # Select a variable without any heuristics.
            for var in self.csp.variables:
                if var not in assignment: return var
        else:
            # Problem 1b
            # Heuristic: most constrained variable (MCV)
            # Select a variable with the least number of remaining domain
values.
            # Hint: given var, self.domains[var] gives you all the possible
values.
            #       Make sure you're finding the domain of the right variable!
            # Hint: get_delta_weight gives the change in weights given a partial
            #       assignment, a variable, and a proposed value to this
variable
            # Hint: for ties, choose the variable with lowest index in
self.csp.variables
            # BEGIN_YOUR_CODE (our solution is 7 lines of code, but don't worry
if you deviate from this)
            mcv = self.csp.variables[0]
            minVal = float('inf')
            for var in self.csp.variables:
                if var not in assignment:
                    sumDeltaWeights = sum(self.get_delta_weight(assignment, var,
x) for x in self.domains[var])
                    if sumDeltaWeights < minVal:
                        minVal = sumDeltaWeights
                        mcv = var
            return mcv
            # END_YOUR_CODE

    def arc_consistency_check(self, var):
        """
        Perform the AC-3 algorithm. The goal is to reduce the size of the
        domain values for the unassigned variables based on arc consistency.

        @param var: The variable whose value has just been set.
        """
        # Problem 1c
        # Hint: How to get variables neighboring variable |var|?
        # => for var2 in self.csp.get_neighbor_vars(var):
        #       # use var2
        #
        # Hint: How to check if a value or two values are inconsistent?
        # - For unary factors
        #   => self.csp.unaryFactors[var1][val1] == 0
        #
        # - For binary factors
        #   => self.csp.binaryFactors[var1][var2][val1][val2] == 0
        #   (self.csp.binaryFactors[var1][var2] returns a nested dict of all
assignments)
        # Hint: Be careful when removing values from lists - trace through
        #       your solution to make sure it behaves as expected.


        # BEGIN_YOUR_CODE (our solution is 15 lines of code, but don't worry if
you deviate from this)
        queue = [var]
        while queue:
            var1 = queue.pop(0)
            for var2 in self.csp.get_neighbor_vars(var1):
                itemsToRemove = []
                for val2 in self.domains[var2]:
```

```
                    consistent = False
                    for val1 in self.domains[var1]:
                        if self.csp.binaryFactors[var1][var2][val1][val2] == 1:
                            consistent = True
                    if self.csp.unaryFactors[var2]:
                        if self.csp.unaryFactors[var2][val2] == 0:
                            consistent = False
                    if not consistent:
                        itemsToRemove.append(val2)
                if itemsToRemove:
                    [self.domains[var2].remove(val2) for val2 in itemsToRemove]
                    queue.append(var2)
        # END_YOUR_CODE


############################################################
# Problem 2b

def get_sum_variable(csp, name, variables, maxSum):
    """
    Given a list of |variables| each with non-negative integer domains,
    returns the name of a new variable with domain range(0, maxSum+1), such that
    it's consistent with the value |n| iff the assignments for |variables|
    sums to |n|.

    @param name: Prefix of all the variables that are going to be added.
        Can be any hashable objects. For every variable |var| added in this
        function, it's recommended to use a naming strategy such as
        ('sum', |name|, |var|) to avoid conflicts with other variable names.
    @param variables: A list of variables that are already in the CSP that
        have non-negative integer values as its domain.
    @param maxSum: An integer indicating the maximum sum value allowed. You
        can use it to get the auxiliary variables' domain

    @return result: The name of a newly created variable with domain range
        [0, maxSum] such that it's consistent with an assignment of |n|
        iff the assignment of |variables| sums to |n|.
    """
    # BEGIN_YOUR_CODE (our solution is 18 lines of code, but don't worry if you
deviate from this)
    result = ('sum', name, 'aggregated')
    if not variables:
        csp.add_variable(result, [0])
        return result
    var = None
    for i, _ in enumerate(variables):
        auxVar = ('sum', name, i)
        if i == 0:
            csp.add_variable(auxVar, [(0, i) for i in range(maxSum + 1)])
            csp.add_binary_factor(auxVar, variables[0], lambda x, y : x[1] == y)
        else:
            csp.add_variable(auxVar, [(j, k) for j in range(maxSum + 1) for k in
range(maxSum + 1)])
            csp.add_binary_factor(auxVar, var, lambda x, y : x[0] == y[1])
            csp.add_binary_factor(auxVar, variables[i], lambda x, y : x[1] ==
(x[0] + y))
        var = auxVar
    csp.add_variable(result, range(maxSum + 1))
    csp.add_binary_factor(result, auxVar, lambda x, y : x == y[1])
    return result
    # END_YOUR_CODE

# importing get_or_variable helper function from util
get_or_variable = util.get_or_variable
```

```
############################################################
# Problem 3

# A class providing methods to generate CSP that can solve the course scheduling
# problem.
class SchedulingCSPConstructor():

    def __init__(self, bulletin, profile):
        """
        Saves the necessary data.

        @param bulletin: Stanford Bulletin that provides a list of courses
        @param profile: A student's profile and requests
        """
        self.bulletin = bulletin
        self.profile = profile

    def add_variables(self, csp):
        """
        Adding the variables into the CSP. Each variable, (request, quarter),
        can take on the value of one of the courses requested in request or
None.
        For instance, for quarter='Aut2013', and a request object, request,
generated
        from 'CS221 or CS246', then (request, quarter) should have the domain
values
        ['CS221', 'CS246', None]. Conceptually, if var is assigned 'CS221'
        then it means we are taking 'CS221' in 'Aut2013'. If it's None, then
        we not taking either of them in 'Aut2013'.

        @param csp: The CSP where the additional constraints will be added to.
        """
        for request in self.profile.requests:
            for quarter in self.profile.quarters:
                csp.add_variable((request, quarter), request.cids + [None])

    def add_bulletin_constraints(self, csp):
        """
        Add the constraints that a course can only be taken if it's offered in
        that quarter.

        @param csp: The CSP where the additional constraints will be added to.
        """
        for request in self.profile.requests:
            for quarter in self.profile.quarters:
                csp.add_unary_factor((request, quarter), \
                    lambda cid: cid is None or \
                        self.bulletin.courses[cid].is_offered_in(quarter))

    def add_norepeating_constraints(self, csp):
        """
        No course can be repeated. Coupling with our problem's constraint that
        only one of a group of requested course can be taken, this implies that
        every request can only be satisfied in at most one quarter.

        @param csp: The CSP where the additional constraints will be added to.
        """
        for request in self.profile.requests:
            for quarter1 in self.profile.quarters:
                for quarter2 in self.profile.quarters:
                    if quarter1 == quarter2: continue
                    csp.add_binary_factor((request, quarter1), (request,
quarter2), \
```

```
                        lambda cid1, cid2: cid1 is None or cid2 is None)

    def get_basic_csp(self):
        """
        Return a CSP that only enforces the basic constraints that a course can
        only be taken when it's offered and that a request can only be satisfied
        in at most one quarter.

        @return csp: A CSP where basic variables and constraints are added.
        """
        csp = util.CSP()
        self.add_variables(csp)
        self.add_bulletin_constraints(csp)
        self.add_norepeating_constraints(csp)
        return csp

    def add_quarter_constraints(self, csp):
        """
        If the profile explicitly wants a request to be satisfied in some given
        quarters, e.g. Aut2013, then add constraints to not allow that request
to
        be satisfied in any other quarter. If a request doesn't specify the
        quarter(s), do nothing.

        @param csp: The CSP where the additional constraints will be added to.
        """
        # Problem 3a
        # Hint: If a request doesn't specify the quarter(s), do nothing.

        # BEGIN_YOUR_CODE (our solution is 5 lines of code, but don't worry if
you deviate from this)
        for request in self.profile.requests:
            if request.quarters:
                for quarter in self.profile.quarters:
                    if quarter not in request.quarters:
                        csp.add_unary_factor((request, quarter), lambda
courseId: courseId == None)
        # END_YOUR_CODE

    def add_request_weights(self, csp):
        """
        Incorporate weights into the CSP. By default, a request has a weight
        value of 1 (already configured in Request). You should only use the
        weight when one of the requested course is in the solution. A
        unsatisfied request should also have a weight value of 1.

        @param csp: The CSP where the additional constraints will be added to.
        """
        for request in self.profile.requests:
            for quarter in self.profile.quarters:
                csp.add_unary_factor((request, quarter), \
                    lambda cid: request.weight if cid != None else 1.0)

    def add_prereq_constraints(self, csp):
        """
        Adding constraints to enforce prerequisite. A course can have multiple
        prerequisites. You can assume that *all courses in req.prereqs are
        being requested*. Note that if our parser inferred that one of your
        requested course has additional prerequisites that are also being
        requested, these courses will be added to req.prereqs. You will be
notified
        with a message when this happens. Also note that req.prereqs apply to
every
        single course in req.cids. If a course C has prerequisite A that is
```

```
requested
        together with another course B (i.e. a request of 'A or B'), then taking
B does
        not count as satisfying the prerequisite of C. You cannot take a course
        in a quarter unless all of its prerequisites have been taken *before*
that
        quarter. You should take advantage of get_or_variable().

        @param csp: The CSP where the additional constraints will be added to.
        """
        # Iterate over all request courses
        for req in self.profile.requests:
            if len(req.prereqs) == 0: continue
            # Iterate over all possible quarters
            for quarter_i, quarter in enumerate(self.profile.quarters):
                # Iterate over all prerequisites of this request
                for pre_cid in req.prereqs:
                    # Find the request with this prerequisite
                    for pre_req in self.profile.requests:
                        if pre_cid not in pre_req.cids: continue
                        # Make sure this prerequisite is taken before the
requested course(s)
                        prereq_vars = [(pre_req, q) \
                            for i, q in enumerate(self.profile.quarters) if i <
quarter_i]
                        v = (req, quarter)
                        orVar = get_or_variable(csp, (v, pre_cid), prereq_vars,
pre_cid)
                        # Note this constraint is enforced only when the course
is taken
                        # in `quarter` (that's why we test `not val`)
                        csp.add_binary_factor(orVar, v, lambda o, val: not val
or o)

    def add_unit_constraints(self, csp):
        """
        Add constraint to the CSP to ensure that the total number of units are
        within profile.minUnits/maxUnits, inclusively. The allowed range for
        each course can be obtained from
bulletin.courses[cid].minUnits/maxUnits.
        For a request 'A or B', if you choose to take A, then you must use a
unit
        number that's within the range of A. You should introduce any additional
        variables that you need. In order for our solution extractor to
        obtain the number of units, for every requested course, you must have
        a variable named (courseId, quarter) (e.g. ('CS221', 'Aut2013')) and
        its assigned value is the number of units.
        You should take advantage of get_sum_variable().

        @param csp: The CSP where the additional constraints will be added to.
        """
        # Problem 3b
        # Hint 1: read the documentation above carefully
        # Hint 2: the domain for each (courseId, quarter) variable should
contain 0
        #         because the course might not be taken
        # Hint 3: add appropriate binary factor between (request, quarter) and
        #         (courseId, quarter) variables
        # Hint 4: you must ensure that the sum of units per quarter for your
schedule
        #         are within the min and max threshold inclusive
        # Hint 5: use nested functions and lambdas like what get_or_variable and
        #         add_prereq_constraints do
        # Hint 6: don't worry about quarter constraints in each Request as
```

```
they'll
        #         be enforced by the constraints added by
add_quarter_constraints

        # BEGIN_YOUR_CODE (our solution is 16 lines of code, but don't worry if
you deviate from this)
        for quarter in self.profile.quarters:
            newVars = []
            for request in self.profile.requests:
                for cid in request.cids:
                    minNumUnits = self.bulletin.courses[cid].minUnits
                    maxNumUnits = self.bulletin.courses[cid].maxUnits
                    domain = list(range(minNumUnits, maxNumUnits + 1)) + [0]
                    csp.add_variable((cid, quarter), domain)
                    newVars.append((cid, quarter))
                    csp.add_binary_factor((request, quarter), (cid, quarter),
lambda course, units: (course == cid and units >= minNumUnits and units <=
maxNumUnits) or (course != cid and units == 0))
            sumVar = get_sum_variable(csp, quarter + 'name', newVars,
self.profile.maxUnits)
            csp.add_unary_factor(sumVar, lambda x: x >= self.profile.minUnits
and x <= self.profile.maxUnits)
        # END_YOUR_CODE

    def add_all_additional_constraints(self, csp):
        """
        Add all additional constraints to the CSP.

        @param csp: The CSP where the additional constraints will be added to.
        """
        self.add_quarter_constraints(csp)
        self.add_request_weights(csp)
        self.add_prereq_constraints(csp)
        self.add_unit_constraints(csp)
```