COSC 76: Artificial Intelligence

Programming Assignment 4: Constraint Satisfaction Problems

Student: Amittai Wekesa (github: @siavava)

Fall 2021

In this programming assignment, you will write a general-purpose constraint solving algorithm, and apply it to solve different CSPs. The learning objective is to implement the methods to solve the CSP, discussed in class. While there can be other previous classes that are useful, the main material covered with this assignment is from 10/4 to 10/8 included, i.e., the PDFs lec10 and lec11.

Getting started

To give you the opportunity to build the code from scratch, this time there is no provided code. If you want, you can take a look at the "Design notes" below, based on our implementation, after the description of the "Required tasks". You are also welcome to come up with your design ideas.

The material in the AIMA book (Chapter 6.1-6.3) and at the class meetings will give you a reference too.

REQUIRED TASKS

1. CSP problem - start with map coloring

Develop a framework that poses the map-coloring problem (as described in the book) as a CSP that the solver can solve.

As a reminder, the map-coloring problem involves several binary constraints. For each pair of adjacent countries, there is a binary constraint that prohibits those countries from having the same color.

CSP Implementation To represent the problem, I built a CSP abstraction that tracks variables, domains, and constraints. It also tracks whether certain heuristics have been enabled or disabled, and if, for instance, debug mode is on or off. The absjtraction also provides a nice interface to outsiders to poll whether a given assignment satisfies the CSP constraints or is a complete assignment.

After some experimentation, I implemented the backtracking algorithm in a separate file. This makes it easier to have the main interface to the algorithm call separate recursive functions depending on whether inferencing is enabled or not.

**Since the code is really long (200+ lines), I did't include it in the report. Check out CSP.py and backtracking.py to see the code.

Results To see the results for this test case, run `test3()` in test_csp.py.

Occassionally, the algorithm finds a solution in 8 calls to `backtrack`, suggesting that it got every variable right on first assignment. However, this doesn't always happen and sometimes the algorithm takes as long as 50 calls! Clearly, there's a lot of inconsistency.

Take 1; solution in 8

```
./test_csp.py

...
CSP
variables:      {'WA', 'NT', 'NSW', 'SA', 'Q', 'T', 'V'}
domains :       {'WA': {'R'}, 'NT': {'G', 'B', 'R'}, 'Q': {'G', 'B', 'R'}, 'NSW': {'G', 'B', 'R'}, 'V':
constraints :   {('SA', 'NSW'), ('WA', 'SA'), ('SA', 'V'), ('NSW', 'V'), ('NT', 'Q'), ('WA', 'NT'), ('S
solution:       {'WA': 'R', 'NT': 'G', 'NSW': 'G', 'SA': 'B', 'Q': 'R', 'T': 'G', 'V': 'R'}
...

         109 function calls (102 primitive calls) in 0.000 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 ./test_csp.py:62(test3)
        1    0.000    0.000    0.000    0.000 erratum.py:37(log_info)
        1    0.000    0.000    0.000    0.000 backtracking.py:18(backtracking_search)
**->  8/1    0.000    0.000    0.000    0.000 backtracking.py:25(backtrack)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.print}
       12    0.000    0.000    0.000    0.000 CSP.py:134(is_consistent)
        1    0.000    0.000    0.000    0.000 CSP.py:54(__str__)
       25    0.000    0.000    0.000    0.000 CSP.py:119(satisfies_constraint)
```

Take 2; Solution in 50

```
./test_csp.py

...
CSP
variables:      {'Q', 'NT', 'NSW', 'SA', 'T', 'V', 'WA'}
domains :       {'WA': {'R'}, 'NT': {'B', 'G', 'R'}, 'Q': {'B', 'G', 'R'}, 'NSW': {'B', 'G', 'R'}, 'V':
constraints :   {('SA', 'NSW'), ('NT', 'Q'), ('WA', 'SA'), ('SA', 'NT'), ('WA', 'NT'), ('SA', 'Q'), ('N
solution:       {'Q': 'R', 'NT': 'B', 'NSW': 'B', 'SA': 'G', 'T': 'B', 'V': 'R', 'WA': R'}
...
```

```
         827 function calls (778 primitive calls) in 0.001 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.001    0.001 ./test_csp.py:62(test3)
        1    0.000    0.000    0.001    0.001 /workspace/personal/python/cs76/PA4/backtracking.py:18(bac
**-> 50/1    0.000    0.000    0.001    0.001 /workspace/personal/python/cs76/PA4/backtracking.py:25(bac
      114    0.000    0.000    0.000    0.000 /workspace/personal/python/cs76/PA4/CSP.py:134(is_consist
      228    0.000    0.000    0.000    0.000 /workspace/personal/python/cs76/PA4/CSP.py:119(satisfies_
        1    0.000    0.000    0.000    0.000 /workspace/personal/python/cs76/PA4/erratum.py:37(log_inf
```

2. Heuristics

Add the heuristics we have seen in class (MRV, degree heuristic, LCV). Make it easy to enable or disable each, since you will want to compare the effectiveness of your results, and since that will aid debugging.

I implemented the Heuristics in a separate heuristics. Since they're implemented as external functions (as opposed to actual code within methods in the CSP class), it's easier to simply call the functions and inspect / debug code specific to the heuristics.

Both MRV heuristic and degree heuristic functions return a single value. However, to enable the use of degree heuristic as a tie-breaker for MRV, I added an optional boolean parameter to my `mrv_heuristic` function that tells the function whether the use of degree heuristic is enabled or not. If enabled, then MRV computes a list of variables with the least remaining value (all having the same value) and passes it to degree heuristic, which checks the degrees of those specific variables and returns the one with the most constraints on other variables. If degree heuristic is not enabled, then MRV returns a single variable with the least remaining values – if multiple variables have the same count, then it will return a random variable from that set of variables.

```
def mrv_heuristic(csp, unassigned: list, deg_heuristic: bool):
    """
        This function implements the MRV (Minimum Remaining Values) heuristic.
        It returns the variable with the fewest legal values.
    """

    # track the variable with least remaining values
    least_var, least_count = None, inf

    # track variables with the same least remaining values
    # this is helpful when using degree heuristic as a tie-breaker.
    tied_vars: list = []

    # loop over unassigned variables
    for var in unassigned:

        # check the size of the domain for the variable.
        count = len(csp.get_domain(var))

        # if size is greater than least size, remember the current variable.
        if count < least_count:
            least_count = count
            least_var = var

            # if degree heuristic is used,
            # clear the list of tied variables.
            # and add the current variable to the list.
            if deg_heuristic: tied_vars = [var]

        # if size is equal to least size,
        # append variable to list of tied variables.
        elif count == least_count and degree_heuristic:
            tied_vars.append(var)

    # if degree heuristic is used,
    # return the variable with the most constraints.
    if deg_heuristic:
        return degree_heuristic(csp, tied_vars)
    # return the variable with the least number of remaining values.
    return least_var
```

On the other hand, LCV (least constraining value) returns an array of values (since that's more efficient than recomputing the values when switching variables). To build this ordered list of values sorted per constraints, it uses a dictionary to check how many constraints each possible value has on other variables (by checking if adjacent variables have the same value in their domains) then sorts the list of values per the number of constraints for each value.

```
# return the domain rearranged by the given restrictions.
sorted_values = sorted(list(domain), key=lambda x: restrictions.get(x, 0))
```

Sample Results

1. No Heuristic

```
        Heuristics enabled:
                MRV: False
                Degree: False
                LCV: False

         1085 function calls (1022 primitive calls) in 0.001 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     64/1    0.000    0.000    0.001    0.001 backtracking.py:23(backtrack)
```

LCV Heuristic By itself, LCV doesn't seem to improve the search that much; it instead makes

```
        Heuristics enabled:
                MRV: False
                Degree: False
                LCV: True

         1682 function calls (1627 primitive calls) in 0.003 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.003    0.003 ./test_csp.py:60(test3)
        1    0.000    0.000    0.003    0.003 backtracking.py:16(backtracking_search)
     56/1    0.000    0.000    0.003    0.003 backtracking.py:23(backtrack)
```

MRV Heuristic Unlike *LCV*, enabling *MRV* heuristic results in massive improvements. The search is still suboptimal and a lot of backtracking, with solutions being found in 16 to 20 calls to the backtrack algorithm.

```
        Heuristics enabled:
                MRV: True
                Degree: False
                LCV: False

         576 function calls (561 primitive calls) in 0.000 seconds

   Ordered by: cumulative time
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 ./test_csp.py:60(test3)
        1    0.000    0.000    0.000    0.000 backtracking.py:16(backtracking_search)
     16/1    0.000    0.000    0.000    0.000 backtracking.py:23(backtrack)
```

Degree Heuristic Degree heuristic also improves the algorithm, with solutions being found in an average of 12 to 18 calls to the backtrack algorithm.

```
        Heuristics enabled:
                MRV: False
                Degree: True
                LCV: False

         576 function calls (565 primitive calls) in 0.000 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 ./test_csp.py:60(test3)
        1    0.000    0.000    0.000    0.000 backtracking.py:16(backtracking_search)
     12/1    0.000    0.000    0.000    0.000 backtracking.py:23(backtrack)
```

All Heuristics Enabled With all three heuristics enabled, the search always finds the assignment in a close to optimal 8 to 10 calls to thfe backtrack algorithm.

```
        Heuristics enabled:
                MRV: True
                Degree: True
                LCV: True

         607 function calls (600 primitive calls) in 0.001 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.001    0.001 ./test_csp.py:60(test3)
        1    0.000    0.000    0.001    0.001 backtracking.py:16(backtracking_search)
      8/1    0.000    0.000    0.001    0.001 backtracking.py:23(backtrack)
```

3. Inference

Add an inference technique (AC-3). Make it easy to enable or disable inference, since you will need to test effectiveness.

I implemented inferencing (AC-3) in inferences.py. The inference functionality modifies the CSP domains.

Because we may later need to undo revisions of the CSP domains, the inferencing function saves a tuple of `(variable, value)` pairs of all edits made. This information is passed back to the scope of the caller, who, in case of failures, reinstates the values in their respective domains.

Revising the Domains

```
# track revisions (we'll potentially need to undo this)
revisions: set = set()

# fetch all from queue.
while queue:

# get variable and a next variable from Queue.
(var, next_var) = queue.remove()

# revise the domain of the variable.
if revise(csp, var, next_var, revisions):

        # get domain
        current_domain = csp.get_domain(var)

        # if domain is empty, CSP is not solvable. Return False.
        if not current_domain:
        return None

        constraints = [c for c in csp.constraints if var in c and next_var not in c]
        for constraint in constraints:
        var1 = constraint[0]
        var2 = constraint[1]
        if var1 == var:
                queue.add((var2, var))
        else:
                queue.add((var1, var))

# return set of revision values (to potentially undo in case the current branch fails).
return revisions
```

Undoing the Revisions Call to undo revisions (in backtracking algorithm):

```
# if no result found, undo the revisions
# and delete the assignment.
if inferencing: undo_revisions(csp, revisions)
```

Function to undo revisions:

```
def undo_revisions(csp: CSP, revisions: set):
    """
        Function to undo domain revisions
    """

    # iterate over all revisions;
    #   reinstate the values in the domains of the respective variables.
    for (variable, value) in revisions:
        csp.get_domain(variable).add(value)
```

Sample Results with inferencing and revisions With inferencing, the search always yields a solution in an optimal 8 calls to backtrack. However, the calls to the inferencing algorithm make the algorithm run 0.004 seconds more.

```
        Heuristics enabled:
                MRV: True
                Degree: True
                LCV: True

         5063 function calls (5056 primitive calls) in 0.004 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.004    0.004 ./test_csp.py:60(test3)
        1    0.000    0.000    0.003    0.003 backtracking.py:16(backtracking_search)
      8/1    0.000    0.000    0.003    0.003 backtracking.py:23(backtrack)
        7    0.001    0.000    0.003    0.000 inferences.py:15(arc_consistency)
```

4. Test on map coloring

You can test your solver with and without heuristics and inference on the map-coloring problem, and describe it in the writeup.

The backtracking algorithm solves the Australian map-coloring problem in less than a second, even without heuristics. However, on closer inspection it immediately becomes clear that the algorithm is inconsistent while running without heuristics, finding solutions in as little as 8 calls (optimal) or as much as 50 calls (very inefficient!).

More information on test runs is incorporated in the discussion above about specific heuristics and inferencing.

5. New problem: Circuit-board layout problem

Write code that describes and solves the circuit-board layout problem. Solve it for at least the example case I have suggested. You should not have to write any more backtracking code; you should be able to use the same implementation you used for map-coloring.

Details about the problem You are given a rectangular circuit board of size n x m, and k rectangular components of arbitrary sizes. Your job is to lay the components out in such a way that they do not overlap. For example, maybe you are given these components:

```
      bbbbb   cc
aaa   bbbbb   cc  eeeeeee
aaa           cc
```

and are asked to lay them out on a 10x3 grid

```
..........
..........
..........
```

A solution might be

```
eeeeeee.cc
aaabbbbbcc
aaabbbbbcc
```

Notice that in this case the solution is not unique!

The variables for this CSP will be the locations of the lower left corner of each component. Assume that the lower left corner of the board has coordinates (0, 0).

Make sure your code displays the output in some nice (enough) way. ASCII art would be fine.

A particularly strong solution might consider several boards, of different sizes and with different numbers, sizes, and shapes of parts.

Discussion points for the write-up In your write-up, describe the domain of a variable corresponding to a component of width w and height h, on a circuit board of width n and height m. Make sure the component fits completely on the board.

Consider components a and b above, on a 10x3 board. In your write-up, write the constraint that enforces the fact that the two components may not overlap. Write out legal pairs of locations explicitly.

Describe how your code converts constraints, etc, to integer values for use by the generic CSP solver.

Implementation Details For the circuit-board layout problem, I implemented a new abstraction, **CircuitProblem**, which inherits the standard **CSP** abstraction and maintains the same interface. For the implementation, see CircuitProblem.py.

The module uses an inner abstraction, **CircuitComponent**, to represent Components that can be placed on the board and moved around. each component has an x and y dimension, and methods that, given a board's dimensions and a specific placement, can compute all the points on the board covered by the component, whether the component overlaps with other given components, etc.

Since no component overlaps with any other component in a consistent assignment, we enforce a constraint for each component against every other component.

NOTE: the circuit board uses IDs to differentiate components, so duplicate components must not have the same string or item as an ID.

To make sure the component fits on the board, each component has a `get_cover()` method that checks if the component fits on the board (and returns a set of all covered spots on the board).

```
def get_cover(self, x, y, board_x, board_y):
    """
        Get the set of all cells covered by the circuit component.
        :arg x: The x coordinate of the circuit component.
        :arg y: The y coordinate of the circuit component.
        :arg board_x: The width of the board.
        :arg board_y: The height of the board.
    """

    # if coordinates are negative, return None
    if x < 0 or y < 0:
        return None

    # if board placement at current position causes overflow,
    #   return None
    if (x + self.dx > board_x) or (y + self.dy > board_y):
        return None

    # loop over all covered positions and add them to a set of all covered positions
    covered = set()
    for i in range(x, x + self.dx):
        for j in range(y, y + self.dy):
            covered.add((i, j))

    # return computed set
    return covered
```

This method is used when checking for valid placements and overlaps, thus ensuring no invalid placement is allowed.

Additionally, the CircuitComponent class has a `get_all_domains()` method that, when called and provided with the parent board's x and y dimensions, computes all valid placements on the board that generate a valid cover for the component. That is:

1. The component does not go out of bounds.
2. Because we are not placing boards yet, we do not need to worry about overlaps.

```python
def get_all_domains(self, board_x, board_y):
    """
        Given a circuit component, and board dimensions,
        return a set of all valid placements for the component on the board.
        :arg board_x: The width of the board.
        :arg board_y: The height of the board.
    """

    # initialize set of all valid placements
    all_domains = set()

    # loop over all cells on the board, test placement.
    # If placement is valid, add it to the set of all valid placements.
    for i in range(board_x):
        for j in range(board_y):
            cover = self.get_cover(i, j, board_x, board_y)
            if cover: all_domains.add((i, j))

    # return set of all valid placements
    return all_domains
```

In your write-up, describe the domain of a variable corresponding to a component of width w and height h, on a circuit board of width n and height m. Make sure the component fits completely on the board.

> The component can be placed at any point (x, y) such that $x + w \leq n$ and $y + h \leq m$. On the other hand, the x and y coordinates must be grater than zero. Therefore, the valid placements are such that $0 < x \leq n - w$ and $0 < y \leq m - h$ respectively.

Each component also has an `overlaps()` method that checks if two circuit components (given their specific placements) do overlaps. This is done by getting the cover for each compoenent, computing the set intersection - in case of no overlap, the intersection must be empty.

```python
def overlaps(self, self_pos, other, other_pos, board_x, board_y):
    """
        Given two circuit components, and their respective positions on the board,
        check whether the two components overlap.
        :arg self_pos: The position of the first circuit component.
        :arg other: The second circuit component.
        :arg other_pos: The position of the second circuit component.
        :arg board_x: The width of the board.
        :arg board_y: The height of the board.
    """

    # Get the cover for each component.
    current_cover = self.get_cover(self_pos[0], self_pos[1], board_x, board_y)
    other_cover = other.get_cover(other_pos[0], other_pos[1], board_x, board_y)

    # if any is empty, overlap impossible.
    if not current_cover or not other_cover:
        return False

    # Find the intersection of the two covers.
    intersection = current_cover & other_cover

    # if the intersection is empty, no overlap.
    return len(intersection) != 0
```

While placing components, we check the placements to ensure no overlapping with previously placed components.

Call from CircuitProblem:

```
# return whether the two components overlap or not.
return not var_1.overlaps(pos_1, var_2, pos_2, self.x, self.y)
```

Consider components a and b above, on a 10x3 board. In your write-up, write the constraint that enforces the fact that the two components may not overlap. Write out legal pairs of locations explicitly.

Legal Pairs:

variable = {("a", 3, 2), ("b", 5, 2)}

NOTE: values represented in a set are each valid with all the assignments for the other variable

```
Valid placements:
a = (1, {1, 2}), b = ({4, 5, 6}, {1, 2});
a = (2, {1, 2}), b = ({5, 6}, {1, 2});
a = (3, {1, 2}), b = (6, {1, 2});
a = (6, {1, 2}), b = (1, {1, 2});
a = (7, {1, 2}), b = ({1, 2}, {1, 2});
a = (8, {1, 2}); b = ({1, 2, 3}, {1, 2});
```

Describe how your code converts constraints to integer values for use with the generic CSP solver.

For the circuit placement problem, I added an `add_variable()` method that, given a tuple representing a component, initializes the circuit component, gets the numerical values for the compoenent and all possible placements, and saves those values into the CSP. We also save in dictionary mappings of each generated value to the original circuit component.

```python
def add_variable(self, var: tuple, domain=None):
    """
            Add a new variable into the CSP.
            variable is a tuple of the form `(id, dx, dy)`
    """

    # Get the variable id, dx, and dy.
    _id = var[0]
    dx = var[1]
    dy = var[2]

    # create new component.
    component = CircuitComponent(_id, dx, dy)

    # add component info into CSP.
    index = len(self.variables)
    self.variable_mappings[index] = component
    self.variables.add(index)
    self.domains.append(set())
    all_values = component.get_all_domains(self.x, self.y)
    for value in all_values:
            if not value in self.value_mappings:
            self.value_mappings.append(value)

            val_index = self.value_mappings.index(value)
            self.domains[-1].add(val_index)


    # add constraint between component and all other components.
    for other_var_index in self.variables:
            if other_var_index != index:
            self.add_constraint(index, other_var_index)

    # increment needed assignments.
    self.needed_assignments += 1
```

Accordingly, once we find a complete placement and want to display it, we reverse the process: find the component matching each numerical variable and find the placement matching every numerical value, and use those to render the result.

```
def __str__(self):

    string = "Circuit Board:\n\n"

    string += "\n".join(self.grid)

    string += "\n\nPlacements:\n\n"

    # if no assignments, return default board
    if not self.assignments:
        string += "\n".join(self.grid)
        return string

    # find all squares to be marked
    all_marked = {}
    for var in self.assignments.keys():
        char = self.variable_mappings[var].id
        assignment = self.value_mappings[self.assignments[var]]
        x, y = assignment[0], assignment[1]
        var = self.variable_mappings[var]
        for ix in range(x, x+var.dx):
            for iy in range(y, y+var.dy):
                all_marked[(ix, iy)] = char

    # build board, marking components where squares are placed.
    for iy in range(self.y):
        for ix in range(self.x):
            string += all_marked.get((ix, iy), ".")
        string += "\n"

    # return string version of board.
    return string
```

Discussion of component placement results

No heuristics, No inferencing Solution found in 6 to 10 calls to the backtrack algorithm.

```
        Heuristics enabled:
                MRV: False
                Degree: False
                LCV: False

         3473 function calls (3464 primitive calls) in 0.002 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.002    0.002 ./test_csp.py:82(test4)
        1    0.000    0.000    0.002    0.002 backtracking.py:16(backtracking_search)
     10/1    0.000    0.000    0.002    0.002 backtracking.py:23(backtrack)
```

Heuristics, No inferencing Solution typically found in exactly 5 calls to the backtrack algorithm (optimal).

```
        Heuristics enabled:
                MRV: True
                Degree: True
                LCV: True

         1470 function calls (1466 primitive calls) in 0.001 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.001    0.001 ./test_csp.py:82(test4)
        1    0.000    0.000    0.000    0.000 /workspace/personal/python/cs76/PA4/backtracking.py:16(bac
      5/1    0.000    0.000    0.000    0.000 /workspace/personal/python/cs76/PA4/backtracking.py:23(bac
```

No Heuristics, Inferencing Enabled Solution typically found in 6 to 8 calls to the backtrack algorithm.

```
        Heuristics enabled:
                MRV: False
                Degree: False
                LCV: False

         60051 function calls (60046 primitive calls) in 0.038 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.038    0.038 ./test_csp.py:82(test4)
        1    0.000    0.000    0.038    0.038 backtracking.py:16(backtracking_search)
      6/1    0.000    0.000    0.038    0.038 backtracking.py:23(backtrack)
```

Heuristics Enabled, Inferencing Enabled Solution typically found in exactly 5 calls to the backtrack algorithm (optimal).

```
        Heuristics enabled:
                MRV: True
                Degree: True
                LCV: True

         43338 function calls (43334 primitive calls) in 0.028 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.028    0.028 ./test_csp.py:82(test4)
        1    0.000    0.000    0.028    0.028 backtracking.py:16(backtracking_search)
      5/1    0.000    0.000    0.028    0.028 backtracking.py:23(backtrack)
```