CSP (Constraint Satisfaction Problem)

Carter Kruse (October 15, 2023)

Instructions

The task for this assignment is to write a general-purpose constraint solving algorithm, and apply it to solve several CSPs. It is useful to be able to build code from scratch. This time, there is no provided code. Make sure you read the detailed design notes below, however. They are based on the professor's solution. here are the objectives.

1. Write a simple backtracking solver.
2. In parallel, develop a framework that poses the map-coloring problem as a CSP that the solver can solve.
3. Test the solver on the map-coloring problem.
4. Write code that describes and solves the circuit-board layout problem. Solve it for at least the example case suggested. You should not have to write any more backtracking code, you should be able to use the same implementation used for map-coloring.
5. Add an inference technique (`MAC-3`). Make it easy to enable or disable inference, since you will need to test effectiveness.
6. Add heuristics (`MRV` , `LCV` , etc). 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.
7. In the write-up, describe the methodology, how you laid out problems, what you implemented for the backtracking search, and compare running time results using different heuristics and inference.
8. (Optional) Implement min-conflicts and apply it to the problems. There are furhter extra credit ideas described at the end of the assignment.

Design Notes

Just like graph search, constraint satisfaction problems can be phrased in a very general way so that a single solver can be applied to different problems. Notice that (CSP) variable names do not really matter in any fundamental way, for example. You could call Western Australia `WA` , or you could call it `0` ; the CSP solver does not care.

So I would recommend that you write some generic CSP class, that solves CSPs with variables that are just numbers. Similarly, values might as well be integers; `1` , `2` , and `3` are equivalent to `r` , `g` , and

`b`. Then an assignment would just be an array of ints. The index into the array would indicate which variable we are talking about, and the value at that location in the array would indicate the value.

Here's an example. For the map-coloring problem, consider the variables `WA`, `NT`, and `V`. Give them indexes `0`, `1`, and `2`, in that order. Consider the values `r`, `g`, and `b`. Let them correspond to integer values `1`, `2`, and `3`. So an assignment is (`WA = g`, `NT = r`, `V = b`). This might be written as

```
assignment[0] = 2;  // WA = g (WA has index 0, g has value 2)
assignment[1] = 1;  // NT = r
assignment[2] = 3;  // V = b
```

So your generic CSP solver just works with integers. A constraint involves some variables, and, given assignments to each of those variables, is either satisfied, or is not. You might have a constraint dictionary that maps from pairs of integers (corresponding to the indices of CSP variables involved in the constraint) to lists of pairs of integers (the list of allowable combinations of values for that pair of variables).

Now your MapColoringCSP would extend your basic CSP class. It would have a constructor that maps from some human-readable description of the particular CSP (maybe just a list of territory names and color domains) into an integer CSP. And then you'd have some output method that takes solutions to the integer CSP, and print them out nicely using territory names and colors. (The constructor might build a hashtable that maps from the integers back to strings for printing.)

Similarly, your circuit-board layout CSP class might do some initial processing to convert variable into integer indices, and values in the domains into integers too.

Map-Coloring Problem

Write a CSP that can solve the map-coloring problem for Australia as described in the book.

Constraints - 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.

The Circuit-Board Layout Problem

You are given a rectangular circuit board of size $ n \times 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 $ 10 \times 3 $ 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) $.

- 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 $ 10 \times 3 $ 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.

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.

CSP

The code for the CSP directly follows from the pseudo-code for the algorithms presented in the book. The backtracking solver, inference technique (`MAC-3`) and heuristics (`MRV` , `LCV` , etc) are implemented in `CSP.py` (in Python). While the file contains quite a few comments, we will quickly review the main components/structure here.

As an overview, this is an implementation of a constraint satisfaction problem (CSP) solver using backtracking search and inference through arc consistency. It includes several enhancements such as minimum remaining values (MRV) and least constraining value (LCV). A bonus min-conflicts search algorithm is also included.

Key Components:

- CSP Class: Represents the CSP with graph, domains, and constraints. Includes methods for checking consistency and detecting completion.
- Backtracking Search: Backtrack function iterates through variables and values using MRV and LCV heuristics.
- Inference: AC-3 queue is used to revise domains and propagate constraints. Helps prune inconsistent values.
- MRV heuristic: Selects next unassigned variable based on smallest domain size. Improves efficiency.
- LCV heuristic: Orders values in domain based on least constraints imposed on future assignments.
- Min-Conflicts Search: Iterative repair method that seeks to reduce conflicts. Useful for highly constrained problems.

Analysis:

- The backtracking search is complete and sound. The heuristics help reduce search tree size.
- AC-3 provides efficient arc consistency propagation. The queues avoid redundant work.
- Min-conflicts can overcome heavy constraints but may fail to find solutions in some cases.

The constructor is responsible for the initialization of the random seed, alongside setting/toggling the inference, MRV, and LCV characteristics of the solution.

```
def __init__(self, inference, MRV, LCV):
    self.inference = inference
    self.MRV = MRV
    self.LCV = LCV
    random.seed(0)
```

The `backtracking_search()` follows from the pseudo-code given in the textbook, and is simply the high-level overview that is used to return a result, starting from an empty assignment.

```
function BACKTRACKING_SEARCH(csp) returns a solution, or failure
    return BACKTRACK({ }, csp)
```

Now, let us consider the main `backtrack()` algorithm, for which the pseudo-code is given in the textbook, as follows. This is responsible for the recursive backtracking, which serves to satisfy all of the constraints of the constraint satisfaction problem.

```
function BACKTRACK(assignment, csp) returns a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        add variable to assignment
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var, assignment)
            if inferences /= failure then
                result ← BACKTRACK(assignment, csp)
                if result /= failure then
                    return result
        remove {var = value} and inferences from assignment
    return failure
```

The backtracking algoirthm for constraint satisfaction problems is modeled on the recursive depth-first search. The functions `SELECT-UNASSIGNED-VARIABLE` and `ORDER-DOMAIN-VALUES`, implement the general-purpose heuristics. The `INFERENCE` function can optionally impose arc-, path-, or k-consistency, as desired. If a value choice leads to failure (noticed either by `INFERENCE` or by `BACKTRACK`), then value assignments (including those made by `INFERENCE`) are retracted and a new value is tried.

The `is_complete()` method is responsible for determining if an assignment is complete or not, specifically if the length of the assignment is equal to the number of elements in the CSP graph.

```
# function IS_COMPLETE(assignment, csp) returns false the assignment is complete and true
def is_complete(self, assignment, csp):
        return len(assignment) == len(csp.graph)
```

The `recover_domain()` method is used to reset the domain, thereby removing inferences from the assignment in the backtracking algorithm. In this sense, we cycle through the variables in the CSP and for those not in the assignment, reset them to the original domain values.

```
# function RECOVER_DOMAIN(assignment, csp) reverts the domain vars not in the assignment
def recover_domain(self, assignment, csp):
```

```
        for var in csp.graph:
            if var not in assignment:
                csp.domain[var] = csp.domain_copy[var]
```

Following this is the inference `AC_3()` algorithm, which again follows closely from the pseudo-code provided in the textbook, given as follows:

```
function AC_3(csp, Yi, assignment) returns false if an inconsistency is found and true ot
    queue ← a queue of arcs, initially all the arcs in csp
    while queue is not empty do
        (Xi, Xj) ← POP(queue)
        if REVISE(csp, Xi, Xj) then
            if size of Di = 0 then return false
            for each Xk in Xi.NEIGHBORS − {Xj} do
                add (Xk, Xi) to queue
    return true
```

After applying `AC_3()` either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The `AC_3()` algorithm relies on the following `revise()` algorithm, for which the pseudo-code from the textbook is as follows:

```
function REVISE(csp, Xi, Xj) returns true iff we revise the domain of Xi
    revised ← false
    for each x in Di do
        if no value y in Dj allows (x, y) to satisfy the constraint between Xi and Xj the
            delete x from Di
            revised ← true
    return revised
```

Now, we transition to the heuristics implemented, which are described as follows. The intuitive idea behind choosing the variable with the fewest "legal" values is called the minimum-remaining-values (`MRV`) heuristic. It also has been called the "most constrained variable" or "fail-first" heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree.

If some variable `X` has no legal values left, the MRV heuristic will select `X` and failure will be detected immediately — avoiding pointless searches through other variables. The MRV heuristic usually performs better than a random or static ordering, sometimes by a factor of 1,000 or more, although the results vary widely depending on the problem.

```
# function MRV_HEURISTIC(assignment, csp) returns unassigned variables, either sorted or
def MRV_heuristic(self, assignment, csp):
    # unassigned_vars ← all variables in the graph that are not in the assignment
```

```
    unassigned_vars = [var for var in csp.graph if var not in assignment]

    if self.MRV:
        # sorted_vars ← unassigned_vars sorted based on minimum remaining values
        sorted_vars = sorted(unassigned_vars, key = lambda var: len(csp.domain[var]))
    else:
        sorted_vars = unassigned_vars

    # return minimum of sorted_vars
    return sorted_vars[0]
```

Similarly, once a variable has been selected, the algorithm must decide on the order in which to examine its values. For this, the least-constraining-value (`LCV`) heuristic can be effective in some cases. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph.

```
# function LCV_HEURISTIC(var, assignment, csp) returns domain, either sorted or not
def LCV_heuristic(self, var, assignment, csp):
    # unsorted_domain ← the domain for var in csp
    unsorted_domain = csp.domain[var]

    if self.LCV:
        # sorted_domain ← unsorted_domain sorted based on least constraining value
        sorted_domain = sorted(unsorted_domain, key = lambda val: self.constraints_impose
    else:
        sorted_domain = unsorted_domain

    # return sorted_domain
    return sorted_domain
```

This LCV heuristic relies on a way to calcluate the number of constraints imposed in order to sort the domain according to the proper order to examine the values. To do so, we consider each variable within the CSP graph (of binary constraints) and determined what the size of the domain would be. This is implemented as follows.

```
# function CONSTRAINTS_IMPOSED(value, var, assignment, csp) returns the number of constra
def constraints_imposed(self, value, var, assignment, csp):
    # constraints ← 0
    constraints = 0

    # for each neighbor in var.NEIGHBORS do
    for neighbor in csp.graph[var]:
        # constraints += the number of constraints imposed
        if neighbor not in assignment:
            constraints += sum([1 for val in csp.domain[neighbor] if val == value])
```

```
    # return constraints
    return constraints
```

The design of the code makes it easy to enable or disable inference and the heuristics (MRV, LCV), which is helfpul to compare the effectiveness of the results, which we will do later.

Map-Coloring Problem

Now let us consider this within the context of the map-coloring problem, which was outlined previously. As stated, our goal is to develop a framework that poses the map-coloring problem as a CSP that the solver can solve, alongside testing the solver on the map-coloring problem. This is implemented in `map_color.py`.

Overview

This program implements a CSP solver to solve the map coloring problem for 7 territories in Australia. It tests different search algorithms (backtracking and min-conflicts) and heuristics (inference, MRV, and LCV). It defines a `map_color` class that sets up the CSP:

- Domain: Color Options [Red, Green, Blue]
- Constraints: Neighboring territories can't have the same color.
- Methods: `is_consistent()`, `show_result()`

The main driver code loops through different algorithm configurations, solves the CSP, prints the results and timing.

Algorithms

- Backtracking search is a systematic depth-first search that backtracks when a variable has no consistent values left.
- Min-conflicts is a local search algorithm that picks the variable with the most conflicts and changes its value to reduce conflicts.

Heuristics

- Inference prunes inconsistent values from neighbor domains when a variable is assigned.
- MRV picks the variable with the smallest remaining domain next.
- LCV picks the value that rules out the fewest choices for the neighbors next.

The struture for the CSP problem set-ups is relatively similar, so we will review this as an example. The code is relatively well-commented, though we aim to review the main components here.

The constructor is responsible for setting up the domain and graph of the CSP, along with the dictionaries used for the final output, to format the results nicely.

```
def __init__(self):
    # set the domain, according to the problem set-up
    self.domain = {}
    for var in range(7):
        self.domain[var] = [1, 2, 3]

    # set the binary constraints in a graph
    self.graph = {0: [1, 2],
                  1: [0, 2, 3],
                  2: [0, 1, 3, 4, 5],
                  3: [1, 2, 4],
                  4: [2, 3, 5],
                  5: [2, 4],
                  6: []}

    # dictionaries used for final output
    self.territory = ['WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T']
    self.color = {1: 'Red', 2: 'Green', 3: 'Blue'}

    self.domain_copy = self.domain
```

The `is_consistent()` method is used in the backtacking search as previously mentioned. In this case, we have a series of binary constraints between bordering territories, indicating that they are not able to be the same color. As such, we check if an arc is consistent by comparing the values of the assignment to the neighboring variables, and the given value.

```
def is_consistent(self, value, var, assignment, csp):
    # cycle through the neighbors of a variable, according to the graph
    for neighbor in csp.graph[var]:
        # check if the colors are the same
        if neighbor in range(len(assignment) - 1) and assignment[neighbor] == value:
            return False

    return True
```

The `show_result()` method simply prints the result using the territory names and the colors, rather than the number format.

```
def show_result(self, assignment):
    # print the result
```

```
        for var in assignment:
            print(str(self.territory[var]) + ': ' + str(self.color[assignment[var]]))
```

Results/Testing

The results of testing, with toggling of the inference, MRV, and LCV are as follows:

```
Inference: True -- MRV: True -- LCV: True

Solution: {0: 1, 1: 2, 2: 3, 3: 1, 4: 2, 5: 1, 6: 1}
(Time: 6.6e-05 Seconds)

WA: Red
NT: Green
SA: Blue
Q: Red
NSW: Green
V: Red
T: Red

Inference: True -- MRV: True -- LCV: False

Solution: {0: 1, 1: 2, 2: 3, 3: 1, 4: 2, 5: 1, 6: 1}
(Time: 2.6e-05 Seconds)

WA: Red
NT: Green
SA: Blue
Q: Red
NSW: Green
V: Red
T: Red

Inference: True -- MRV: False -- LCV: True

Solution: {0: 1, 1: 2, 2: 3, 3: 1, 4: 2, 5: 1, 6: 1}
(Time: 3.29e-05 Seconds)

WA: Red
NT: Green
SA: Blue
Q: Red
NSW: Green
V: Red
T: Red

Inference: True -- MRV: False -- LCV: False
```

```
Solution: {0: 1, 1: 2, 2: 3, 3: 1, 4: 2, 5: 1, 6: 1}
(Time: 2.1e-05 Seconds)

WA: Red
NT: Green
SA: Blue
Q: Red
NSW: Green
V: Red
T: Red

Inference: False -- MRV: True -- LCV: True

Solution: {0: 1, 1: 2, 2: 3, 3: 1, 4: 2, 5: 1, 6: 1}
(Time: 2.88e-05 Seconds)

WA: Red
NT: Green
SA: Blue
Q: Red
NSW: Green
V: Red
T: Red

Inference: False -- MRV: True -- LCV: False

Solution: {0: 1, 1: 2, 2: 3, 3: 1, 4: 2, 5: 1, 6: 1}
(Time: 1.69e-05 Seconds)

WA: Red
NT: Green
SA: Blue
Q: Red
NSW: Green
V: Red
T: Red

Inference: False -- MRV: False -- LCV: True

Solution: {0: 1, 1: 2, 2: 3, 3: 1, 4: 2, 5: 1, 6: 1}
(Time: 2.48e-05 Seconds)

WA: Red
NT: Green
SA: Blue
Q: Red
NSW: Green
V: Red
T: Red
```

```
Inference: False -- MRV: False -- LCV: False

Solution: {0: 1, 1: 2, 2: 3, 3: 1, 4: 2, 5: 1, 6: 1}
(Time: 1.31e-05 Seconds)

WA: Red
NT: Green
SA: Blue
Q: Red
NSW: Green
V: Red
T: Red
```

In this sense, the code effectively implements and tests different CSP solving algorithms and heuristics on the map coloring problem. The results demonstrate that there is a tradeoff between optimality and speed, though for a simple problem like this, there is little difference in the runtime. Typically, we find the following, given that this CSP is relatively small:

- Inference, MRV, and LCV increase the time complexity of the algorithm, indicating that the program takes slightly longer to run.
- Regardless, for problems that may be slightly more complex, inference and the associated heuristics may improve the amount of time it takes to reach a solution, given that they are responsible for arc consistency (pruning) and reordering.
- The path to the solution is heavily dependent on the problem, indicating that an analysis of the problem is required before trying to implement inference or the heurisitics.

In this case, given the nature of the problem and its simplicity, the inference and heuristics do not yield a different solution to the CSP, despite the fact that there are multiple solutions. We will see that this is not always the case, particularly for the circuit-board problem.

Circuit Board Problem

Now let us consider this within the context of the circuit-board problem, which was outlined previously. As stated, our goal is to develop a framework that describes and solves the circuit-board layout problem, solving it for at least the example case suggested. There is no need to write any more backtracking code, as we are able to use the same implementation used for map-coloring. This is implemented in `circuit_board.py` .

Overview:

This code implements a CSP solver to arrange circuit board components on a board without overlapping. It utilizes backtracking search along with inference, minimum remaining values (MRV), and least constraining value (LCV) heuristics.

Implementation Details

- The `circuit_board` class represents the CSP, which initializes the board dimensions, components, locations domain, and constraint graph.
- The `is_consistent()` method checks if a variable assignment leads to overlapping components.
- The CSP class conducts the backtracking search. It has inference, MRV, and LCV parameters to toggle the heuristics on/off.
- Main execution tests different combinations of the heuristics and prints the runtimes and solutions.

Discussion Questions

The variables for this CSP are the locations of the lower left corner of each component. We assume that the lower left corner of the board has coordinates $ (0, 0) $.

As requested, we now 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 $. The domain is represented as a location/coordinate $ (x, y) $, where $ 0 <= x < n - w + 1 $ and $ 0 <= y < m - h + 1 $. This allows the component to fit completely on the board, as the location is bounded appropriately.

Now, we consider components `a` and `b` as given, on a $ 10 \times 3 $ board. The constraint that enforces the fact that the two components may not overlap would typically be given by a mathematical expression of inequality, though in this implementation, we use a different configuration that allows for non-rectangular components, as described below. Nonetheless, we may still write out legal pairs of locations explicity, as if we were solving the problem by hand, as follows:

B Coordinate	A Coordinate
(0, 0) or (0, 1)	[(5, 0), (5, 1), (6, 0), (6, 1), (7, 0), (7, 1)]
(1, 0) or (1, 1)	[(6, 0), (6, 1), (7, 0), (7, 1)]
(2, 0) or (2, 1)	[(7, 0), (7, 1)]
(3, 0) or (3, 1)	[(0, 0), (0, 1)]
(4, 0) or (4, 1)	[(0, 0), (1, 1), (1, 0), (1, 1)]
(5, 0) or (5, 1)	[(0, 0), (1, 1), (1, 0), (1, 1), (2, 0), (2, 1)]

In this case, we first express the possible coordinates of the `b` component, and then express the possible locations of the smaller `a` component which satisfy non-overlapping

Further, the code converts constraints, etc, to integer values for use by the generic CSP solver by assigning each component to an index $ i $, which has a domain that represents a list of

locations/points on the circuit board. The constraints are simply represented by a fully connected graph between all of the components, and the overlapping is handled through the `is_consistent()` method..

The struture for the CSP problem set-ups is relatively similar, so we will review this as an example. The code is relatively well-commented, though we aim to review the main components here.

The constructor is responsible for setting up the domain and graph of the CSP. The width and height of the circuit board, along with the components are set during this phase. To determine the domain, we must cycle through the possible x, y-locations for each component, according to it's width and height.

For the graph, we simply add binary constraints between all components, as we do not want any of them to overlap with one another. This differs from the map-coloring problem, though is relatively straightforward.

```
def __init__(self, n, m, components):
    # set the width, height, and components
    self.width = n
    self.height = m
    self.components = components

    # set the domain, according to the problem set-up
    self.domain = {}
    for var in range(len(components)):
        # initialize an empty list of locations
        locations = []

        # cycle through the possible x, y locations
        for x in range(self.width - len(self.components[var][0]) + 1):
            for y in range(self.height - len(self.components[var]) + 1):
                locations.append((x, y))

        self.domain[var] = locations

    # set the binary constraints in a graph
    self.graph = {}
    for i in range(len(components)):
        self.graph[i] = [j for j in range(len(components)) if j != i]

    self.domain_copy = self.domain
```

The `is_consistent()` method is responsible for detecting if there is an overlap in components. Typically, this is accomplished by comparing the locations of the components (alongside the bounding

values determined by the width/length), though the code is modified to allow non-rectangular components, so this method will not work.

Instead, we consider each location on the circuit board (as a grid), and mark the number of times that a location is filled by part of a component. In this way, we are able to easily detect if two components overlap, by simply incrementing the number of "layers" over any given space, and determining if the number of "layers" is greater than 1.

```
  def is_consistent(self, value, var, assignment, csp):
      # create a grid to keep track of overlapping components
      grid = [[0 for _ in range(self.width)] for _ in range(self.height)]

      # cycle through the variables in assignment
      for variable in assignment:
          # extract the x, y coordinates
          x_start, y_start = assignment[variable]

          # cycle through the x, y locations
          for y in range(y_start, y_start + len(self.components[variable])):
              for x in range(x_start, x_start + len(self.components[variable][0])):
                  # determine the character at a given location
                  character = self.components[variable][y - y_start][x - x_start]

                  # handle the case where a character is not the empty space
                  if character != '.':
                      grid[self.height - y - 1][x] += 1

                      if grid[self.height - y - 1][x] > 1:
                          return False

      return True
```

The `show_result()` method is simply responsible for displaying the circuit board will all of the components placed on the board. By cycling through the width and height of the board, we are able to place every component. In this case, the dispay is slightly more complicated, as we need to search the values of assignment at a given location to determine the appropriate component piece to place at a given square, though the code follows similarly to the method for checking if there is an overlap. As indicated by the project instructions, the code displays the output using ASCII art, which is nice enough.

```
  def show_result(self, assignment):
      # print the result
      grid = []

      # cycle through the height, width
```

```
        for _ in range(self.height):
            row = []
            for _ in range(self.width):
                row.append('.')

            grid.append(row)

        # # # # #

        # cycle through the variables in assignment
        for variable in assignment:
            # extract the x, y coordinates
            x_start, y_start = assignment[variable]

            # cycle through the x, y locations
            for y in range(y_start, y_start + len(self.components[variable])):
                for x in range(x_start, x_start + len(self.components[variable][0])):
                    # determine the character at a given location
                    character = self.components[variable][y - y_start][x - x_start]

                    # handle the case where a character is not the empty space
                    if character != '.':
                        grid[self.height - y - 1][x] = character

        # cycle through the height, width
        for j in range(self.height):
            for i in range(self.width):
                # display the grid
                print(grid[j][i], end = '')
            print()
```

Results/Testing

The results of testing, with toggling of the inference, MRV, and LCV are as follows. In this case, we consider the basic case presented by the problem assignment, though will delve into different configurations later. As indicated, a particularly strong solution might consider several boards, of different sizes and with different numbers, sizes, and shapes of parts, which is what this program does to merit extra credit, as defined later.

```
Inference: True -- MRV: True -- LCV: True

Solution: {2: (8, 0), 1: (0, 0), 3: (0, 2), 0: (5, 0)}
(Time: 0.000406 Seconds)

eeeeeee.cc
bbbbbaaacc
bbbbbaaacc
```

```
Inference: True -- MRV: True -- LCV: False

Solution: {2: (0, 0), 1: (2, 0), 3: (2, 2), 0: (7, 0)}
(Time: 0.000435 Seconds)

cceeeeeee.
ccbbbbbaaa
ccbbbbbaaa

Inference: True -- MRV: False -- LCV: True

Solution: {0: (7, 1), 1: (2, 1), 2: (0, 0), 3: (2, 0)}
(Time: 0.00154 Seconds)

ccbbbbbaaa
ccbbbbbaaa
cceeeeeee.

Inference: True -- MRV: False -- LCV: False

Solution: {0: (0, 0), 1: (3, 0), 2: (8, 0), 3: (0, 2)}
(Time: 0.000268 Seconds)

eeeeeee.cc
aaabbbbbcc
aaabbbbbcc

Inference: False -- MRV: True -- LCV: True

Solution: {2: (8, 0), 1: (0, 0), 3: (0, 2), 0: (5, 0)}
(Time: 0.00031 Seconds)

eeeeeee.cc
bbbbbaaacc
bbbbbaaacc

Inference: False -- MRV: True -- LCV: False

Solution: {2: (0, 0), 1: (2, 0), 3: (2, 2), 0: (7, 0)}
(Time: 0.00038 Seconds)

cceeeeeee.
ccbbbbbaaa
ccbbbbbaaa

Inference: False -- MRV: False -- LCV: True

Solution: {0: (7, 1), 1: (2, 1), 2: (0, 0), 3: (2, 0)}
```

```
(Time: 0.00133 Seconds)

ccbbbbbaaa
ccbbbbbaaa
cceeeeeee.

Inference: False -- MRV: False -- LCV: False

Solution: {0: (0, 0), 1: (3, 0), 2: (8, 0), 3: (0, 2)}
(Time: 0.000189 Seconds)

eeeeeee.cc
aaabbbbbcc
aaabbbbbcc
```

As previously mentioned, with a slightly more complex problem, we find that the solutions are not unique, and they are determined by the inference (toggled on/off) and the heuristics used. This means that for an underconstrained problem, there may be many solutions to the CSP, which the CSP solver addresses differently depending on the order of the domain, inference, etc.

Analysis

Now, let us perform a quick analysis on the runtimes.

- Inference appears to reduce runtimes, likely by pruning inconsistent values from domains earlier.
- MRV has mixed impact, sometimes increasing and sometimes decreasing runtimes. It helps guide search but has overhead of computing MRV.
- LCV also has mixed impact, but tends to increase runtimes. It reduces constraints but has overhead of computing LCV.

Thus, we find that combining inference, MRV, and LCV gives good performance by leveraging the strengths of each heuristic, though it comes at the cost of slightly more computational time.

Extra Credit/Bonus

The following implementations are designed to enhance the problem, thus meriting "extra credit" as defined by the outlines of the assignment. By going above and beyond to consider extensions to the problem, we consider a greater level of complexity for the problem. Due to the structure of our code, we are able to maintain much of the structure of the previously written code. The following represent implementation of a few of the ideas that were encouraged.

USA - Bonus #1

Similar to the map-coloring problem for Australia, we may implement a 4-color map problem for the United States, defining the domain and constraints in a similar fashion to that of the `map_color.py` file. In this instance, however, it is better to represent the graph of the constraints in a separate file, which we read in to the Python script upon execution. This significantly cleans up the code and allows us to focus on the parts that are more important.

The dictionary containing the CSP graph, which represents the constraints, is given in `usa.txt` . The Python script that models the 4-coloring of the USA map is given in `usa.py` . While the structure of the code mostly follows from the `map_color.py` file, we will review the differences here.

The differences are highlighted in the `init()` method, which is responsible for the creation of the domain, according to the problem set up. Now, we have 4 colors, represented by integers 1 through 4, as opposed to 3, which would not provide a solution for the coloring of the United States. Further, as stated, we implement a way to read in the data from a file, with the appropriate error checking. The territories are updated to represent state abbreviations, while the color yellow is added to the list of colors.

```
def __init__(self):
    # set the domain, according to the problem set-up
    self.domain = {}
    for var in range(51):
        self.domain[var] = [1, 2, 3, 4]

    # open the file
    with open('usa.txt', 'r') as file:
        # read the contents
        file_content = file.read()

        try:
            # set the binary constraints in a graph
            self.graph = ast.literal_eval(file_content)

            # handle the case where the graph is not a dictionary
            if not isinstance(self.graph, dict):
                self.graph = None
                raise ValueError('The content of the file is not a dictionary.')

        # handle the case where there is an error loading the data
        except (SyntaxError, ValueError) as error:
            self.graph = None
            print('Error loading data. ' + str(error))

    # dictionaries used for final output
    self.state = ['AK', 'AL', 'AR', 'AZ', 'CA', 'CO', 'CT', 'DC', 'DE', 'FL', 'GA', 'HI',
    self.color = {1: 'Red', 2: 'Green', 3: 'Blue', 4: 'Yellow'}
```

```
        self.domain_copy = self.domain
```

The main reason for the implementation of this code was to further test the significance of inference and the various heurisitics used in the main algorithm. The previous iterations of map coloring and the circuit board are not sufficiently complex to develop a true analysis of the runtime, given the speed of the program.

While this code took slightly longer to run, it was still relativlely quick, considering the number of constraints that were placed. This highlights the robustness of the algorithm implemented, particularly with the inference and heuristics. Thus, we present the results here, alongside an analysis, for the more advanced problem.

```
Inference: True -- MRV: True -- LCV: True

Solution: {0: 1, 1: 1, 2: 1, 3: 1, 4: 2, 5: 2, 6: 1, 7: 1, 8: 1, 9: 2, 10: 3, 11: 1, 12:
(Time: 56.5 Seconds)

AK: Red
AL: Red
AR: Red
AZ: Red
CA: Green
CO: Green
CT: Red
DC: Red
DE: Red
FL: Green
GA: Blue
HI: Red
IA: Red
ID: Red
IL: Green
IN: Red
KS: Red
KY: Blue
LA: Blue
MA: Green
MD: Green
ME: Red
MI: Green
MN: Blue
MO: Yellow
MS: Yellow
MT: Blue
NC: Red
ND: Red
```

```
NE: Blue
NH: Blue
NJ: Green
NM: Yellow
NV: Yellow
NY: Yellow
OH: Yellow
OK: Blue
OR: Blue
PA: Blue
RI: Blue
SC: Green
SD: Green
TN: Green
TX: Green
UT: Blue
VA: Yellow
VT: Red
WA: Green
WI: Yellow
WV: Red
WY: Yellow

Inference: True -- MRV: True -- LCV: False

Solution: {0: 1, 1: 1, 2: 1, 3: 1, 4: 2, 5: 2, 6: 1, 7: 1, 8: 1, 9: 2, 10: 3, 11: 1, 12:
(Time: 47.7 Seconds)

AK: Red
AL: Red
AR: Red
AZ: Red
CA: Green
CO: Green
CT: Red
DC: Red
DE: Red
FL: Green
GA: Blue
HI: Red
IA: Red
ID: Red
IL: Green
IN: Red
KS: Red
KY: Blue
LA: Blue
MA: Green
MD: Green
```

```
ME: Red
MI: Green
MN: Blue
MO: Yellow
MS: Yellow
MT: Blue
NC: Red
ND: Red
NE: Blue
NH: Blue
NJ: Green
NM: Yellow
NV: Yellow
NY: Yellow
OH: Yellow
OK: Blue
OR: Blue
PA: Blue
RI: Blue
SC: Green
SD: Green
TN: Green
TX: Green
UT: Blue
VA: Yellow
VT: Red
WA: Green
WI: Yellow
WV: Red
WY: Yellow

Inference: True -- MRV: False -- LCV: True

Solution: {0: 1, 1: 1, 2: 1, 3: 1, 4: 2, 5: 2, 6: 1, 7: 1, 8: 1, 9: 2, 10: 3, 11: 1, 12:
(Time: 52.6 Seconds)

AK: Red
AL: Red
AR: Red
AZ: Red
CA: Green
CO: Green
CT: Red
DC: Red
DE: Red
FL: Green
GA: Blue
HI: Red
IA: Red
```

```
ID: Red
IL: Green
IN: Red
KS: Red
KY: Blue
LA: Blue
MA: Green
MD: Green
ME: Red
MI: Green
MN: Blue
MO: Yellow
MS: Yellow
MT: Blue
NC: Red
ND: Red
NE: Blue
NH: Blue
NJ: Green
NM: Yellow
NV: Yellow
NY: Yellow
OH: Yellow
OK: Blue
OR: Blue
PA: Blue
RI: Blue
SC: Green
SD: Green
TN: Green
TX: Green
UT: Blue
VA: Yellow
VT: Red
WA: Green
WI: Yellow
WV: Red
WY: Yellow

Inference: True -- MRV: False -- LCV: False

Solution: {0: 1, 1: 1, 2: 1, 3: 1, 4: 2, 5: 2, 6: 1, 7: 1, 8: 1, 9: 2, 10: 3, 11: 1, 12:
(Time: 43.8 Seconds)

AK: Red
AL: Red
AR: Red
AZ: Red
CA: Green
```

```
CO: Green
CT: Red
DC: Red
DE: Red
FL: Green
GA: Blue
HI: Red
IA: Red
ID: Red
IL: Green
IN: Red
KS: Red
KY: Blue
LA: Blue
MA: Green
MD: Green
ME: Red
MI: Green
MN: Blue
MO: Yellow
MS: Yellow
MT: Blue
NC: Red
ND: Red
NE: Blue
NH: Blue
NJ: Green
NM: Yellow
NV: Yellow
NY: Yellow
OH: Yellow
OK: Blue
OR: Blue
PA: Blue
RI: Blue
SC: Green
SD: Green
TN: Green
TX: Green
UT: Blue
VA: Yellow
VT: Red
WA: Green
WI: Yellow
WV: Red
WY: Yellow

Inference: False -- MRV: True -- LCV: True
```

```
Solution: {0: 1, 1: 1, 2: 1, 3: 1, 4: 2, 5: 2, 6: 1, 7: 1, 8: 1, 9: 2, 10: 3, 11: 1, 12:
(Time: 51.7 Seconds)

AK: Red
AL: Red
AR: Red
AZ: Red
CA: Green
CO: Green
CT: Red
DC: Red
DE: Red
FL: Green
GA: Blue
HI: Red
IA: Red
ID: Red
IL: Green
IN: Red
KS: Red
KY: Blue
LA: Blue
MA: Green
MD: Green
ME: Red
MI: Green
MN: Blue
MO: Yellow
MS: Yellow
MT: Blue
NC: Red
ND: Red
NE: Blue
NH: Blue
NJ: Green
NM: Yellow
NV: Yellow
NY: Yellow
OH: Yellow
OK: Blue
OR: Blue
PA: Blue
RI: Blue
SC: Green
SD: Green
TN: Green
TX: Green
UT: Blue
VA: Yellow
```

```
VT: Red
WA: Green
WI: Yellow
WV: Red
WY: Yellow

Inference: False -- MRV: True -- LCV: False

Solution: {0: 1, 1: 1, 2: 1, 3: 1, 4: 2, 5: 2, 6: 1, 7: 1, 8: 1, 9: 2, 10: 3, 11: 1, 12:
(Time: 43.1 Seconds)

AK: Red
AL: Red
AR: Red
AZ: Red
CA: Green
CO: Green
CT: Red
DC: Red
DE: Red
FL: Green
GA: Blue
HI: Red
IA: Red
ID: Red
IL: Green
IN: Red
KS: Red
KY: Blue
LA: Blue
MA: Green
MD: Green
ME: Red
MI: Green
MN: Blue
MO: Yellow
MS: Yellow
MT: Blue
NC: Red
ND: Red
NE: Blue
NH: Blue
NJ: Green
NM: Yellow
NV: Yellow
NY: Yellow
OH: Yellow
OK: Blue
OR: Blue
```

```
PA: Blue
RI: Blue
SC: Green
SD: Green
TN: Green
TX: Green
UT: Blue
VA: Yellow
VT: Red
WA: Green
WI: Yellow
WV: Red
WY: Yellow

Inference: False -- MRV: False -- LCV: True

Solution: {0: 1, 1: 1, 2: 1, 3: 1, 4: 2, 5: 2, 6: 1, 7: 1, 8: 1, 9: 2, 10: 3, 11: 1, 12:
(Time: 47.6 Seconds)

AK: Red
AL: Red
AR: Red
AZ: Red
CA: Green
CO: Green
CT: Red
DC: Red
DE: Red
FL: Green
GA: Blue
HI: Red
IA: Red
ID: Red
IL: Green
IN: Red
KS: Red
KY: Blue
LA: Blue
MA: Green
MD: Green
ME: Red
MI: Green
MN: Blue
MO: Yellow
MS: Yellow
MT: Blue
NC: Red
ND: Red
NE: Blue
```

```
NH: Blue
NJ: Green
NM: Yellow
NV: Yellow
NY: Yellow
OH: Yellow
OK: Blue
OR: Blue
PA: Blue
RI: Blue
SC: Green
SD: Green
TN: Green
TX: Green
UT: Blue
VA: Yellow
VT: Red
WA: Green
WI: Yellow
WV: Red
WY: Yellow

Inference: False -- MRV: False -- LCV: False

Solution: {0: 1, 1: 1, 2: 1, 3: 1, 4: 2, 5: 2, 6: 1, 7: 1, 8: 1, 9: 2, 10: 3, 11: 1, 12:
(Time: 39.1 Seconds)

AK: Red
AL: Red
AR: Red
AZ: Red
CA: Green
CO: Green
CT: Red
DC: Red
DE: Red
FL: Green
GA: Blue
HI: Red
IA: Red
ID: Red
IL: Green
IN: Red
KS: Red
KY: Blue
LA: Blue
MA: Green
MD: Green
ME: Red
```

```
MI: Green
MN: Blue
MO: Yellow
MS: Yellow
MT: Blue
NC: Red
ND: Red
NE: Blue
NH: Blue
NJ: Green
NM: Yellow
NV: Yellow
NY: Yellow
OH: Yellow
OK: Blue
OR: Blue
PA: Blue
RI: Blue
SC: Green
SD: Green
TN: Green
TX: Green
UT: Blue
VA: Yellow
VT: Red
WA: Green
WI: Yellow
WV: Red
WY: Yellow
```

Overall, we see that implementing inference does typically improve performance, though the added time complexity of MRV and LCV slows down the solution significantly, despite the fact that the minimum remaining values and least constrained value are designed to enhance the solution. We propose that this may not be the case due to the nature of the problem... there are relatively few constraints to consider, as the graph of states is not fully connected. The degree of any given vertex is no greater than three, so it is relatively easy to find a solution without considering the heuristics.

Non-Rectangular Components (Circuit Board) - Bonus #2

Let us return to the circuit board problem, specifically modifying the circuit board model to allow non-rectangular components. This extension was expressed in the assignment, and while difficult, was relatively straightforward to implement with the proper adjustment.

As indicated in the discussion of the circuit board problem, the original idea of considering overlapping components by their "bounding boxes" was no longer valid, so a different method had to be developed in order to produce a solution. This involved cycling through the x, y-locations and keeping

track of the number of "layers", essentially how many components were on a location, there were. In this sense, the domain remains the same, as we only consider the x, y-locations for which a given component will fit. To represent the non-rectangular components, we still must have a "bounding box" in this sense, though the components may be filled with empty space, which allows us to construct these unique components.

As before the initialization allows for specification of the domain, according to the problem set up, and the binary constraints in the CSP graph, which will be used later to enforce the non-overlapping condition.

```
def __init__(self, n, m, components):
    # set the width, height, and components
    self.width = n
    self.height = m
    self.components = components

    # set the domain, according to the problem set-up
    self.domain = {}
    for var in range(len(components)):
        # initialize an empty list of locations
        locations = []

        # cycle through the possible x, y locations
        for x in range(self.width - len(self.components[var][0]) + 1):
            for y in range(self.height - len(self.components[var]) + 1):
                locations.append((x, y))

        self.domain[var] = locations

    # set the binary constraints in a graph
    self.graph = {}
    for i in range(len(components)):
        self.graph[i] = [j for j in range(len(components)) if j != i]

    self.domain_copy = self.domain
```

The `is_consistent()` method is responsible for making sure that components do not overlap. For this, we use the "layering" method as previously described, which cycles through a grid and counts the number of components that are at any given x, y-location. If there is more than a single component placed at an x, y-location, we return false, as the value of the variable is not consistent with the assignment.

```
def is_consistent(self, value, var, assignment, csp):
    # create a grid to keep track of overlapping components
    grid = [[0 for _ in range(self.width)] for _ in range(self.height)]
```

```
        # cycle through the variables in assignment
        for variable in assignment:
            # extract the x, y coordinates
            x_start, y_start = assignment[variable]

            # cycle through the x, y locations
            for y in range(y_start, y_start + len(self.components[variable])):
                for x in range(x_start, x_start + len(self.components[variable][0])):
                    # determine the character at a given location
                    character = self.components[variable][y - y_start][x - x_start]

                    # handle the case where a character is not the empty space
                    if character != '.':
                        grid[self.height - y - 1][x] += 1

                        if grid[self.height - y - 1][x] > 1:
                            return False

        return True
```

Finally, the `show_result()` method must be modified to show the result, according to the non-rectangular components. To do so, we intialize the empty grid with all empty spaces, and then cycle through the variables in the assignment, keeping track of where we are for a particular variable. By cycling through the x, y-locations, we determine which character is at a given point and update the grid as long as the character is not an empty space.

```
    def show_result(self, assignment):
        # print the result
        grid = []

        # cycle through the height, width
        for _ in range(self.height):
            row = []
            for _ in range(self.width):
                row.append('.')

            grid.append(row)

        # # # # #

        # cycle through the variables in assignment
        for variable in assignment:
            # extract the x, y coordinates
            x_start, y_start = assignment[variable]

            # cycle through the x, y locations
```

```
            for y in range(y_start, y_start + len(self.components[variable])):
                for x in range(x_start, x_start + len(self.components[variable][0])):
                    # determine the character at a given location
                    character = self.components[variable][y - y_start][x - x_start]

                    # handle the case where a character is not the empty space
                    if character != '.':
                        grid[self.height - y - 1][x] = character

        # cycle through the height, width
        for j in range(self.height):
            for i in range(self.width):
                # display the grid
                print(grid[j][i], end = '')
            print()
```

Now, we may use various components that are non-rectangular, alongside considering the variation of the board to be different sizes. This emphasizes the implementation of an extremely versatile, strong solution, and thus is deserving of extra credit. The following indicate a few testing cases for various configurations, which highlight the speed of the program as well.

The following demonstrates the highly underconstrained case, in which the circuit board is expanded with the same number of pieces. Evidently, the solution converges very quickly.

```
components = [['aaa', 'aaa'],
              ['bbbbb', 'bbbbb'],
              ['cc', 'cc', 'cc'],
              ['eeeeeee']]
n, m = 14, 4



Inference: True -- MRV: True -- LCV: True

Solution: {2: (12, 0), 1: (0, 0), 3: (0, 3), 0: (5, 0)}
(Time: 0.000914 Seconds)

eeeeeee.......
............cc
bbbbbaaa....cc
bbbbbaaa....cc

Inference: True -- MRV: True -- LCV: False

Solution: {2: (0, 0), 1: (2, 0), 3: (0, 3), 0: (7, 0)}
(Time: 0.000739 Seconds)

eeeeeee.......
```

```
cc...........
ccbbbbbaaa....
ccbbbbbaaa....

Inference: True -- MRV: False -- LCV: True

Solution: {0: (10, 2), 1: (0, 2), 2: (8, 0), 3: (0, 0)}
(Time: 0.000611 Seconds)

bbbbb.....aaa.
bbbbb...ccaaa.
........cc....
eeeeeee.cc....

Inference: True -- MRV: False -- LCV: False

Solution: {0: (0, 0), 1: (0, 2), 2: (5, 0), 3: (5, 3)}
(Time: 0.000808 Seconds)

bbbbbeeeeeee..
bbbbbcc.......
aaa..cc.......
aaa..cc.......

Inference: False -- MRV: True -- LCV: True

Solution: {2: (12, 0), 1: (0, 0), 3: (0, 3), 0: (5, 0)}
(Time: 0.000653 Seconds)

eeeeeee.......
............cc
bbbbbaaa....cc
bbbbbaaa....cc

Inference: False -- MRV: True -- LCV: False

Solution: {2: (0, 0), 1: (2, 0), 3: (0, 3), 0: (7, 0)}
(Time: 0.000482 Seconds)

eeeeeee.......
cc...........
ccbbbbbaaa....
ccbbbbbaaa....

Inference: False -- MRV: False -- LCV: True

Solution: {0: (10, 2), 1: (0, 2), 2: (8, 0), 3: (0, 0)}
(Time: 0.000365 Seconds)
```

```
bbbbb.....aaa.
bbbbb...ccaaa.
........cc....
eeeeeee.cc....

Inference: False -- MRV: False -- LCV: False

Solution: {0: (0, 0), 1: (0, 2), 2: (5, 0), 3: (5, 3)}
(Time: 0.000556 Seconds)

bbbbbeeeeeee..
bbbbbcc.......
aaa..cc.......
aaa..cc.......
```

The following demonstrates the case where an extra component is added, and the `a` component is no longer rectangular. Despite this, the solution fits all of the pieces on the board, and it converges to a solution rather quickly.

```
components = [['aaa', 'aa.'],
              ['bbbbb', 'bbbbb'],
              ['cc', 'cc', 'cc'],
              ['eeeeeee'],
              ['d', 'd']]
n, m = 10, 3


Inference: True -- MRV: True -- LCV: True

Solution: {2: (8, 0), 1: (0, 0), 3: (0, 2), 0: (5, 0), 4: (7, 1)}
(Time: 0.000794 Seconds)

eeeeeeedcc
bbbbbaadcc
bbbbbaaacc

Inference: True -- MRV: True -- LCV: False

Solution: {2: (0, 0), 1: (2, 0), 3: (2, 2), 0: (7, 0), 4: (9, 1)}
(Time: 0.000845 Seconds)

cceeeeeeed
ccbbbbbaad
ccbbbbbaaa

Inference: True -- MRV: False -- LCV: True
```

```
Solution: {0: (7, 0), 1: (2, 0), 2: (0, 0), 3: (2, 2), 4: (9, 1)}
(Time: 0.00818 Seconds)

cceeeeeeed
ccbbbbbaad
ccbbbbbaaa

Inference: True -- MRV: False -- LCV: False

Solution: {0: (5, 0), 1: (0, 0), 2: (8, 0), 3: (0, 2), 4: (7, 1)}
(Time: 0.0101 Seconds)

eeeeeeedcc
bbbbbaadcc
bbbbbaaacc

Inference: False -- MRV: True -- LCV: True

Solution: {2: (8, 0), 1: (0, 0), 3: (0, 2), 0: (5, 0), 4: (7, 1)}
(Time: 0.000496 Seconds)

eeeeeeedcc
bbbbbaadcc
bbbbbaaacc

Inference: False -- MRV: True -- LCV: False

Solution: {2: (0, 0), 1: (2, 0), 3: (2, 2), 0: (7, 0), 4: (9, 1)}
(Time: 0.000549 Seconds)

cceeeeeeed
ccbbbbbaad
ccbbbbbaaa

Inference: False -- MRV: False -- LCV: True

Solution: {0: (7, 0), 1: (2, 0), 2: (0, 0), 3: (2, 2), 4: (9, 1)}
(Time: 0.0053 Seconds)

cceeeeeeed
ccbbbbbaad
ccbbbbbaaa

Inference: False -- MRV: False -- LCV: False

Solution: {0: (5, 0), 1: (0, 0), 2: (8, 0), 3: (0, 2), 4: (7, 1)}
(Time: 0.00713 Seconds)

eeeeeeedcc
```

```
bbbbbaadcc
bbbbbaaacc
```

The following demonstrates a different case where a non-rectangular component is added (and L shape), and the `b` and `c` components are no longer rectangular. Once again, the solution fits all of the pieces on the board, and it converges to a solution rather quickly. This particular case would be difficult to determine by hand quickly, as the lack of rectangular pieces makes it more like a puzzle than previous iterations.

```
components = [['aaa', 'aaa'],
              ['bbbbb', 'bbbb.'],
              ['cc', '.c', 'cc'],
              ['eeeeeee'],
              ['dd', 'd.']]
n, m = 10, 3
```

```
Inference: True -- MRV: True -- LCV: True

Solution: {2: (8, 0), 1: (3, 0), 3: (0, 2), 0: (0, 0), 4: (7, 1)}
(Time: 0.00847 Seconds)

eeeeeeedcc
aaabbbbddc
aaabbbbbcc

Inference: True -- MRV: True -- LCV: False

Solution: {2: (8, 0), 1: (3, 0), 3: (0, 2), 0: (0, 0), 4: (7, 1)}
(Time: 0.0185 Seconds)

eeeeeeedcc
aaabbbbddc
aaabbbbbcc

Inference: True -- MRV: False -- LCV: True

Solution: {0: (0, 0), 1: (3, 0), 2: (8, 0), 3: (0, 2), 4: (7, 1)}
(Time: 0.0149 Seconds)

eeeeeeedcc
aaabbbbddc
aaabbbbbcc

Inference: True -- MRV: False -- LCV: False

Solution: {0: (0, 0), 1: (3, 0), 2: (8, 0), 3: (0, 2), 4: (7, 1)}
```

```
(Time: 0.000378 Seconds)

eeeeeeedcc
aaabbbbddc
aaabbbbbcc

Inference: False -- MRV: True -- LCV: True

Solution: {2: (8, 0), 1: (3, 0), 3: (0, 2), 0: (0, 0), 4: (7, 1)}
(Time: 0.00461 Seconds)

eeeeeeedcc
aaabbbbddc
aaabbbbbcc

Inference: False -- MRV: True -- LCV: False

Solution: {2: (8, 0), 1: (3, 0), 3: (0, 2), 0: (0, 0), 4: (7, 1)}
(Time: 0.0116 Seconds)

eeeeeeedcc
aaabbbbddc
aaabbbbbcc

Inference: False -- MRV: False -- LCV: True

Solution: {0: (0, 0), 1: (3, 0), 2: (8, 0), 3: (0, 2), 4: (7, 1)}
(Time: 0.011 Seconds)

eeeeeeedcc
aaabbbbddc
aaabbbbbcc

Inference: False -- MRV: False -- LCV: False

Solution: {0: (0, 0), 1: (3, 0), 2: (8, 0), 3: (0, 2), 4: (7, 1)}
(Time: 0.00028 Seconds)

eeeeeeedcc
aaabbbbddc
aaabbbbbcc
```

N Queens - Bonus #3

The following extension was expressed in the assignment. Apply your CSP solver to solve an interesting problem of your choosing. In this case, the N queens problem is implemented as a CSP, as

it is well-represented by this type of model, and thus allows for the problem to be solved without modification to the CSP code.

Similar to the map-coloring and circuit board problems, we may implement the N queens problem, defining the domain and constraints in a similar fashion, yet dependent on the problem in particular. In this instance, the graph of the constraints for the CSP is represented as a fully connected graph, as with the circuit board problem, which allows us to handle the constraints in the `is_consistent()` method, which we will explain later.

The `init()` method defines the size of the chessboard, along with the number of queens, determining the domain of each of the queens as the locations on the chessboard $ (i, j) $. Following this, the binary constraints are set in the graph.

```
def __init__(self, n):
    # set the number of queens
    self.n = n

    # set the domain, according to the problem set-up
    self.domain = {}
    for var in range(self.n):
        self.domain[var] = [(i, j) for i in range(self.n) for j in range(self.n)]

    # set the binary constraints in a graph
    self.graph = {}
    for i in range(self.n):
        self.graph[i] = [j for j in range(self.n) if j != i]

    self.domain_copy = self.domain
```

The `is_consistent()` method is relatively similar to that for the map-coloring problem, although this time, we not only consider the case where the assignment values are equal (indicating the same position on the chessboard, or same color in the case of map-coloring), but further all of the positions on the chessboard that are "under attack" from the queens that are already placed on the chessboard, according to the assignment.

```
def is_consistent(self, value, var, assignment, csp):
    # cycle through the neighbors of a variable, according to the graph
    for neighbor in csp.graph[var]:
        # check if the position of a queen is not appropriate
        if neighbor in range(len(assignment) - 1) and any(val == value for val in self.at
            return False

    return True
```

To do so, we must construct all of the positions on a chessboard of size $ n \times n $ that are attacked by a queen at a given position, so the verticals, horizontals, and diagonals. This code could be further adopted for any chess piece, including knights (theoretically) to solve a different problem. To determine the attacked locations, first we generate a list of potential locations, and then sort out those that are outside the bounds of the chessboard.

```
  def attacked_positions(self, value):
      # create a set of potential positions
      potential = set()

      # extract the x, y coordinates
      x, y = value

      # horizontal, vertical constraints
      for i in range(self.n):
          potential.add((x, i))
          potential.add((i, y))

      # diagonal constraints
      for i in range(self.n):
          potential.add((x - i, y - i))
          potential.add((x - i, y + i))
          potential.add((x + i, y - i))
          potential.add((x + i, y + i))

      # create a list of attacked (valid) locations
      attacked = []

      # cycle through the positions
      for position in potential:
          # check the bounds of the position, according to the chessboard
          if position[0] >= 0 and position[0] < self.n and position[1] >= 0 and position[1]
              attacked.append(position)

      return attacked
```

Finally, we must modify the `show_result()` code to be particular to this problem. To do so, we cycle over the columns and rows of the chessboard, which allows us to index each square on the board, and then determine if there is a queen at that location. Due to the symmetry of the problem, solutions may be reflected along vertical and horizontal axes without loss of generality.

```
  def show_result(self, assignment):
      # print the result
      for j in range(self.n):
          for i in range(self.n):
              if (i, j) in assignment.values():
```

```
                print('Q', end = '')
            else:
                print('.', end = '')
        print()
```

The following shows the result for the 4 queens problem, which is solved rather quickly by the CSP. In this case, we consider the impact of inference, MRV, and LCV, but the algorithm runs so quickly that it makes a minor contribution.

```
Inference: True -- MRV: True -- LCV: True

Solution: {0: (0, 1), 1: (1, 3), 2: (2, 0), 3: (3, 2)}
(Time: 0.00337 Seconds)

..Q.
Q...
...Q
.Q..

Inference: True -- MRV: True -- LCV: False

Solution: {0: (0, 1), 1: (1, 3), 2: (2, 0), 3: (3, 2)}
(Time: 0.00297 Seconds)

..Q.
Q...
...Q
.Q..

Inference: True -- MRV: False -- LCV: True

Solution: {0: (0, 1), 1: (1, 3), 2: (2, 0), 3: (3, 2)}
(Time: 0.00354 Seconds)

..Q.
Q...
...Q
.Q..

Inference: True -- MRV: False -- LCV: False

Solution: {0: (0, 1), 1: (1, 3), 2: (2, 0), 3: (3, 2)}
(Time: 0.00263 Seconds)

..Q.
Q...
...Q
```

```
.Q..

Inference: False -- MRV: True -- LCV: True

Solution: {0: (0, 1), 1: (1, 3), 2: (2, 0), 3: (3, 2)}
(Time: 0.00577 Seconds)

..Q.
Q...
...Q
.Q..

Inference: False -- MRV: True -- LCV: False

Solution: {0: (0, 1), 1: (1, 3), 2: (2, 0), 3: (3, 2)}
(Time: 0.00228 Seconds)

..Q.
Q...
...Q
.Q..

Inference: False -- MRV: False -- LCV: True

Solution: {0: (0, 1), 1: (1, 3), 2: (2, 0), 3: (3, 2)}
(Time: 0.00234 Seconds)

..Q.
Q...
...Q
.Q..

Inference: False -- MRV: False -- LCV: False

Solution: {0: (0, 1), 1: (1, 3), 2: (2, 0), 3: (3, 2)}
(Time: 0.00212 Seconds)

..Q.
Q...
...Q
.Q..
```

Similarily, the following are the results for the 5 queens and 6 queens problem, both of which are solved quickly by the CSP, indicating the robustness of the implementation.

```
Inference: True -- MRV: True -- LCV: True

Solution: {0: (0, 0), 1: (1, 2), 2: (2, 4), 3: (3, 1), 4: (4, 3)}
```

```
(Time: 0.0016 Seconds)

Q....
...Q.
.Q...
....Q
..Q..

Inference: True -- MRV: True -- LCV: False

Solution: {0: (0, 0), 1: (1, 2), 2: (2, 4), 3: (3, 1), 4: (4, 3)}
(Time: 0.00122 Seconds)

Q....
...Q.
.Q...
....Q
..Q..

Inference: True -- MRV: False -- LCV: True

Solution: {0: (0, 0), 1: (1, 2), 2: (2, 4), 3: (3, 1), 4: (4, 3)}
(Time: 0.00155 Seconds)

Q....
...Q.
.Q...
....Q
..Q..

Inference: True -- MRV: False -- LCV: False

Solution: {0: (0, 0), 1: (1, 2), 2: (2, 4), 3: (3, 1), 4: (4, 3)}
(Time: 0.00126 Seconds)

Q....
...Q.
.Q...
....Q
..Q..

Inference: False -- MRV: True -- LCV: True

Solution: {0: (0, 0), 1: (1, 2), 2: (2, 4), 3: (3, 1), 4: (4, 3)}
(Time: 0.00144 Seconds)

Q....
...Q.
.Q...
```

```
....Q
..Q..

Inference: False -- MRV: True -- LCV: False

Solution: {0: (0, 0), 1: (1, 2), 2: (2, 4), 3: (3, 1), 4: (4, 3)}
(Time: 0.000863 Seconds)

Q....
...Q.
.Q...
....Q
..Q..

Inference: False -- MRV: False -- LCV: True

Solution: {0: (0, 0), 1: (1, 2), 2: (2, 4), 3: (3, 1), 4: (4, 3)}
(Time: 0.00117 Seconds)

Q....
...Q.
.Q...
....Q
..Q..

Inference: False -- MRV: False -- LCV: False

Solution: {0: (0, 0), 1: (1, 2), 2: (2, 4), 3: (3, 1), 4: (4, 3)}
(Time: 0.000827 Seconds)

Q....
...Q.
.Q...
....Q
..Q..


Inference: True -- MRV: True -- LCV: True

Solution: {0: (0, 1), 1: (1, 3), 2: (2, 5), 3: (3, 0), 4: (4, 2), 5: (5, 4)}
(Time: 0.694 Seconds)

...Q..
Q.....
....Q.
.Q....
.....Q
..Q...
```

```
Inference: True -- MRV: True -- LCV: False

Solution: {0: (0, 1), 1: (1, 3), 2: (2, 5), 3: (3, 0), 4: (4, 2), 5: (5, 4)}
(Time: 0.634 Seconds)

...Q..
Q.....
....Q.
.Q....
.....Q
..Q...

Inference: True -- MRV: False -- LCV: True

Solution: {0: (0, 1), 1: (1, 3), 2: (2, 5), 3: (3, 0), 4: (4, 2), 5: (5, 4)}
(Time: 0.672 Seconds)

...Q..
Q.....
....Q.
.Q....
.....Q
..Q...

Inference: True -- MRV: False -- LCV: False

Solution: {0: (0, 1), 1: (1, 3), 2: (2, 5), 3: (3, 0), 4: (4, 2), 5: (5, 4)}
(Time: 0.633 Seconds)

...Q..
Q.....
....Q.
.Q....
.....Q
..Q...

Inference: False -- MRV: True -- LCV: True

Solution: {0: (0, 1), 1: (1, 3), 2: (2, 5), 3: (3, 0), 4: (4, 2), 5: (5, 4)}
(Time: 0.608 Seconds)

...Q..
Q.....
....Q.
.Q....
.....Q
..Q...

Inference: False -- MRV: True -- LCV: False
```

```
Solution: {0: (0, 1), 1: (1, 3), 2: (2, 5), 3: (3, 0), 4: (4, 2), 5: (5, 4)}
(Time: 0.564 Seconds)

...Q..
Q.....
....Q.
.Q....
.....Q
..Q...

Inference: False -- MRV: False -- LCV: True

Solution: {0: (0, 1), 1: (1, 3), 2: (2, 5), 3: (3, 0), 4: (4, 2), 5: (5, 4)}
(Time: 0.605 Seconds)

...Q..
Q.....
....Q.
.Q....
.....Q
..Q...

Inference: False -- MRV: False -- LCV: False

Solution: {0: (0, 1), 1: (1, 3), 2: (2, 5), 3: (3, 0), 4: (4, 2), 5: (5, 4)}
(Time: 0.565 Seconds)

...Q..
Q.....
....Q.
.Q....
.....Q
..Q...
```

Min-Conflicts - Bonus #4

A further extension that was mentioned in the assignment was the following. Implement min-conflicts and apply it to the problems. In the following, we specifically apply a min-conflicts local search algorithm to determine the solution to the Australia map problem.

As an overview the min-conflicts algorithm is used for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The `conflicts()` function counts the number of constraints violated by a particular value, given the rest of the current assignment.

The code for the min-conflicts algorithm directly follows from the pseudo-code for the algorithms presented in the book, and is implemented in the `CSP.py` file (in Python), close to the end. While the file contains quite a few comments, we will quickly review the main components/structure here.

The `min_conflicts_search()` method implements the pseudo-code that is presented in the textbook, which is gien as follows. According to a maximum number of steps, the algorithm searches for a local solution, by initializing a complete assignment, choosing conflicted variables, and minimizing the number of conflicts for a given value, updating the variable with the appropriate value.

```
function MIN_CONFLICTS_SEARCH(csp, max_steps) returns a solution, or failure
    current ← an initial complete assignment for csp
    for i = 1 to max_steps do
        if current is a solution for csp then return current
        var ← a randomly chosen conflicted variable from csp.VARIABLES
        value ← the value v for var that minimizes CONFLICTS(var, current, csp)
        set var = value in current
    return failure
```

As implied, this search algorithm is dependent on other methods, namely `initialize_assignment()`, `choose_conflicted_variable()`, and `min_conflicts_value()`, alongside `conflicts()`.

The `initialize_assignment()` method is responsible for creating a random complete assignment for the CSP, which may or may not satisfy the constraints that are specified. However, it does satisfy the unary constraints on the domain.

```
# function INITIALIZE_ASSIGNMENT(csp) returns a complete assignment for csp
def initialize_assignment(self, csp):
    # assignment ← an empty assignment for csp
    assignment = {}

    # for each var in csp.VARIABLES
    for var in csp.graph:
        # assign a random value in var.DOMAIN to var
        assignment[var] = random.choice(csp.domain[var])

    # return assignment
    return assignment
```

The `choose_conflicted_variable()` method is responsible for returning a random conflicted variable, that is, one that makes it so that the CSP is not arc consistent. In this sense, the code is relatively straightforward, yet relevant.

```
# function CHOOSE_CONFLICTED_VARIABLE(assignment, csp) returns a conflicted variable
def choose_conflicted_variable(self, assignment, csp):
    # conflicted_vars ← a list of vars in csp.VARIABLES where the value is not consistent
    conflicted_vars = [var for var in csp.graph if not csp.is_consistent(assignment[var],

    # return random choice of conflicted_vars
    return random.choice(conflicted_vars)
```

The `min_conflicts_value()` method is responsible for returning the value that minimizes the number of conflicts in the CSP for a given variable. To do so, we use the LCV heuristic to order the domain values, and then sort the values according to the number of conflicts.

```
# function MIN_CONFLICTS_VALUE(var, assignment, csp) returns the the value that minimizes
def min_conflicts_value(self, var, assignment, csp):
    # values ← list[value] in ORDER-DOMAIN-VALUES(var, assignment, csp)
    values = self.LCV_heuristic(var, assignment, csp)

    # sorted_values ← values sorted based on conflicts
    sorted_values = sorted(values, key = lambda val: self.conflicts(var, val, assignment,

    # return minimum of sorted_vars
    return sorted_values[0]
```

Finally, the `conflicts()` method is responsible for counting the number of conflicts that a given value assigned to a variable creates, given the assignment in the context of the CSP. To do so, we assign the variable to the value in the assignment, check the conflicts, and then remove the variable from the assignment, which mirrors the code from before.

```
# function CONFLICTS(var, value, assignment, csp) returns the number of conflicts
def conflicts(self, var, value, assignment, csp):
    # values ← list[value] in ORDER-DOMAIN-VALUES(var, assignment, csp)
    conflicts = 0

    # add variable to assignment
    assignment[var] = value

    # conflicts += the number of conflicts for neighbor in var.NEIGHBORS
    conflicts += sum([1 for neighbor in csp.graph[var] if neighbor in assignment and not

    # remove {var = value}
    del assignment[var]

    # return conflicts
    return conflicts
```

This serves to create a local search algorithm that is different than the backtracking search algorithm. Typically, local search algorithms are used in problems where we are trying to minimize the number of conflicted values, particularly situations in which there may not be an entirely perfect solution, or in the case where we have already reached a solution and there is a change to the constraints.

Rather than re-run the backtracking search, we are able to use a local search algorithm to find the nearest solution that satisfies the constraints, as opposed to having to start all over again. This increases the effectiveness of re-optimizing for a new set of slightly-changed constraints.

When we apply the min-conflicts search algorithm to the map coloring problem, we converge on a solution rather quickly, as an initial assignment of colors to the territories is already relatively close to the correct solution. This would not be the case for the USA map, as the assignment of colors may vary drastically from the correct assignment, due to the fact that the constraints make the problem more restrictive.

The following show the results for the Australia map-coloring problem when running the min-conflicts search algorithm.

```
Inference: True -- MRV: True -- LCV: True

Solution: {0: 2, 2: 1, 3: 2, 4: 3, 5: 2, 6: 2, 1: 3}
(Time: 4.32e-05 Seconds)

WA: Green
SA: Red
Q: Green
NSW: Blue
V: Green
T: Green
NT: Blue

Inference: True -- MRV: True -- LCV: False

Solution: {0: 2, 2: 1, 3: 2, 4: 3, 5: 2, 6: 2, 1: 3}
(Time: 2.88e-05 Seconds)

WA: Green
SA: Red
Q: Green
NSW: Blue
V: Green
T: Green
NT: Blue

Inference: True -- MRV: False -- LCV: True
```

```
Solution: {0: 2, 2: 1, 3: 2, 4: 3, 5: 2, 6: 2, 1: 3}
(Time: 2.84e-05 Seconds)

WA: Green
SA: Red
Q: Green
NSW: Blue
V: Green
T: Green
NT: Blue

Inference: True -- MRV: False -- LCV: False

Solution: {0: 2, 2: 1, 3: 2, 4: 3, 5: 2, 6: 2, 1: 3}
(Time: 2.6e-05 Seconds)

WA: Green
SA: Red
Q: Green
NSW: Blue
V: Green
T: Green
NT: Blue

Inference: False -- MRV: True -- LCV: True

Solution: {0: 2, 2: 1, 3: 2, 4: 3, 5: 2, 6: 2, 1: 3}
(Time: 2.79e-05 Seconds)

WA: Green
SA: Red
Q: Green
NSW: Blue
V: Green
T: Green
NT: Blue

Inference: False -- MRV: True -- LCV: False

Solution: {0: 2, 2: 1, 3: 2, 4: 3, 5: 2, 6: 2, 1: 3}
(Time: 2.67e-05 Seconds)

WA: Green
SA: Red
Q: Green
NSW: Blue
V: Green
T: Green
NT: Blue
```

```
Inference: False -- MRV: False -- LCV: True

Solution: {0: 2, 2: 1, 3: 2, 4: 3, 5: 2, 6: 2, 1: 3}
(Time: 2.72e-05 Seconds)

WA: Green
SA: Red
Q: Green
NSW: Blue
V: Green
T: Green
NT: Blue

Inference: False -- MRV: False -- LCV: False

Solution: {0: 2, 2: 1, 3: 2, 4: 3, 5: 2, 6: 2, 1: 3}
(Time: 2.57e-05 Seconds)

WA: Green
SA: Red
Q: Green
NSW: Blue
V: Green
T: Green
NT: Blue
```

To run the min-conflicts search algorithm, please uncomment the associated line in the `map_color.py` file.

Literature Review - Bonus #5

The map-coloring problem from this assignment is a well-known one, and it has been studied (relatively) extensively in the AI reserach community (alongside problems involving n queens and other games). While there is no no required additional part of the assignment, I found it relevant to complete a brief review of a somewhat-relate paper CSP problems.

Thus, a paper was chosen and read through enough to get a sense of the approach and major findings. In this report, we briefly discuss the paper. The discussion should describe the problem attacked by the paper, give a quick summary of the main result(s), and discuss the basic approach of the paper.

Paper: *An Optimal Coarse-Grained Arc Consistency Algorithm* by Christian Bessière, Jean-Charles Régin, Roland H.C. Yap, Yuanlin Zhang

This paper provides an overview of a new arc consistency algorithm for constraint satisfaction problems that operates between multiple variables at a time (coarse-grained), rather than constraints between individual variables (fine-grained). The main result is the proof that this algorithm is optimal, by propagating arc consistency in the minimum amount of time compared to other coarse-grained algorithms in the worst case.

In constraint satisfaction problems (CSPs), arc consistency is a crucial concept that refers to the extent to which constraints in a CSP are satisfied. Coarse-grained arc consistency typically involves checking and enforcing consistency across larger groups of variables rather than individual pairs, making it more efficient in certain scenarios.

Abstract "The use of constraint propagation is the main feature of any constraint solver. It is thus of prime importance to manage the propagation in an efficient and effective fashion. There are two classes of propagation algorithms for general constraints: fine-grained algorithms where the removal of a value for a variable will be propagated to the corresponding values for other variables, and coarse-grained algorithms where the removal of a value will be propagated to the related variables. One big advantage of coarse-grained algorithms, like AC-3, over fine-grained algorithms, like AC-4, is the ease of integration when implementing an algorithm in a constraint solver."

The key idea is that fine-grained algorithms typically have the optimal worst case time complexity, while coarse-grained algorithms do not, indicating a tradeoff. AC-3, the algorithm used for inference for this assignment, has a non-optimal worst case complexity, despite it being relatively simple and efficient, and thus widely used.

The authors allude to the idea that a course-grained algorithm, AC2001/3.1, is worst case optimal, yet preserves as much as possible the ease of integration. The idea may be applied to obtain a path consistency algorithm (instead of arc consistency) that has the best-known time and space complexity, which is further extended to non-binary constraints.

In summary, the paper addresses an approach to achieving optimal coarse-grained arc consistency, which has various implications for CSPs. This allows algorithms to effectively detect inconsistencies (especially in the worst case) and prune the search space across larger subsets of variables at once compared to regular fine-grained arc consistency algorithms.

CS1 Sections - Bonus #6

The following provides an indication of an extension, specifically the CS 1 section assignment problem. The provided information is as follows:

At the beginning of every term in which CS 1 is offered, we need to assign students in the class into different sections. There are section leaders, and there are students. Each student and section leader fills out an availability form. We'll simplify things for now, and assume people are "available" or "not available" for a given time, and not worry about "prefer to avoid" times.

Then you get a pair of long lists of comma-separated values: one for section leaders and one for students.

```
Boba Fett, Monday 4:00, Monday 5:00, Monday 6:00, Tuesday 7:00, Wednesday 12:30
Chewbacca, Tuesday 6:00, Wednesday 4:00
```

Student names (and section leader names) are the variables. You might want to prepend an asterisk to section leader names to make it clear which is which in debugging. Values are section times. Domains for each variables are specified by the lists.

For test purposes, you will probably want to write a program that generates a few lists like this randomly, and saved them each in a text file. Student names like "Student1" would be fine. Then your CSP can load that text file.

The goal is to find an assignment of students to sections that satisfies several constraints:

- There is exactly one section leader in each section (an `alldiff` constraint).
- There is no student in a section that does not also contain a section leader. (Notice that there are more available section times on the form than there will be sections.)
    - This is a global constraint, and will need to be specially implemented.
- If there are $ n $ students and $ k $ section leaders, then no section has fewer than $ n / k - 1 $ students, and no section has more than $ n / k + 1 $ students.

You do not need an explicit constraint to specifiy that a student can only meet at specified times - that's part of the domain already. (You can think of the domain as a unuary constraint.)

There are probably more- and less- efficient ways of implementing these global constraints. The easiest way is probably to use a partial assignment to build a set for each section, and test those sets for particular properties. You can test for the properties as the sets are constructed, for early termination.

While the following implementation does not fully cover all of these aspects, it provides at least a start to solving the CS 1 section assignment. In this case, we generate a random list of times that the section leaders are available for, and impose the constraint that no two section leaders may be in the same section.

In this instance, the graph of the constraints for the CSP is represented as a fully connected graph, as with the circuit board problem, which allows us to handle the constraints in the `is_consistent()` method, which we will explain later.

The `init()` method defines the section times and generates a list of 5 section leaders, with a random sampling of the times, even with a random number of available times. Following this, the binary constraints are set in the graph.

```
  def __init__(self):
      # initialize the random seed
      random.seed(0)

      # set the times and leaders
      self.times = ['M 4:00', 'M 5:00', 'M 6:00', 'T 7:00', 'W 12:30', 'F 5:00']
      self.leaders = [['Leader' + str(i)] + random.sample(self.times, random.randint(1, ler

      # set the domain, according to the problem set-up
      self.domain = {}
      for leader in self.leaders:
          self.domain[leader[0]] = leader[1:]

      # set the binary constraints in a graph
      self.graph = {}
      for i in self.domain:
          self.graph[i] = [j for j in self.domain if j != i]

      self.domain_copy = self.domain
```

The `is_consistent()` method is basically identical to that for the map-coloring problem, although this time, the colors are not being compared, rather, it is is the available times. We do not require that this be mapped to an integer value.

```
  def is_consistent(self, value, var, assignment, csp):
      # cycle through the neighbors of a variable, according to the graph
      for neighbor in csp.graph[var]:
          # check if the times are the same
          if neighbor in assignment and assignment[neighbor] == value:
              return False

      return True
```

Finally, we must modify the `show_result()` code to be particular to this problem. To do so, we simply output the results, according the assignment, which provides the section times that the section leaders should be assigned to in order to meet all of the constraints.

```
  def show_result(self, assignment):
      # print the result
```

```
    for var in assignment:
        print(str(var) + ': ' + str(assignment[var]))
```

The following shows the result for the CS 1 section assignment problem, which is solved rather quickly by the CSP. In this case, we consider the impact of inference, MRV, and LCV, but the algorithm runs so quickly that it makes a minor contribution.

```
Inference: True -- MRV: True -- LCV: True

Solution: {'Leader3': 'W 12:30', 'Leader4': 'M 5:00', 'Leader5': 'M 4:00', 'Leader0': 'M
(Time: 8.49e-05 Seconds)

Leader3: W 12:30
Leader4: M 5:00
Leader5: M 4:00
Leader0: M 6:00
Leader1: T 7:00
Leader2: F 5:00

Inference: True -- MRV: True -- LCV: False

Solution: {'Leader3': 'W 12:30', 'Leader4': 'F 5:00', 'Leader5': 'M 4:00', 'Leader0': 'T
(Time: 3.29e-05 Seconds)

Leader3: W 12:30
Leader4: F 5:00
Leader5: M 4:00
Leader0: T 7:00
Leader1: M 6:00
Leader2: M 5:00

Inference: True -- MRV: False -- LCV: True

Solution: {'Leader0': 'T 7:00', 'Leader1': 'M 6:00', 'Leader2': 'M 5:00', 'Leader3': 'W 1
(Time: 6.22e-05 Seconds)

Leader0: T 7:00
Leader1: M 6:00
Leader2: M 5:00
Leader3: W 12:30
Leader4: F 5:00
Leader5: M 4:00

Inference: True -- MRV: False -- LCV: False

Solution: {'Leader0': 'T 7:00', 'Leader1': 'M 6:00', 'Leader2': 'M 5:00', 'Leader3': 'W 1
(Time: 3.29e-05 Seconds)
```

```
Leader0: T 7:00
Leader1: M 6:00
Leader2: M 5:00
Leader3: W 12:30
Leader4: F 5:00
Leader5: M 4:00

Inference: False -- MRV: True -- LCV: True

Solution: {'Leader3': 'W 12:30', 'Leader4': 'M 5:00', 'Leader5': 'M 4:00', 'Leader0': 'M
(Time: 3.89e-05 Seconds)

Leader3: W 12:30
Leader4: M 5:00
Leader5: M 4:00
Leader0: M 6:00
Leader1: T 7:00
Leader2: F 5:00

Inference: False -- MRV: True -- LCV: False

Solution: {'Leader3': 'W 12:30', 'Leader4': 'F 5:00', 'Leader5': 'M 4:00', 'Leader0': 'T
(Time: 1.5e-05 Seconds)

Leader3: W 12:30
Leader4: F 5:00
Leader5: M 4:00
Leader0: T 7:00
Leader1: M 6:00
Leader2: M 5:00

Inference: False -- MRV: False -- LCV: True

Solution: {'Leader0': 'T 7:00', 'Leader1': 'M 6:00', 'Leader2': 'M 5:00', 'Leader3': 'W 1
(Time: 3.81e-05 Seconds)

Leader0: T 7:00
Leader1: M 6:00
Leader2: M 5:00
Leader3: W 12:30
Leader4: F 5:00
Leader5: M 4:00

Inference: False -- MRV: False -- LCV: False

Solution: {'Leader0': 'T 7:00', 'Leader1': 'M 6:00', 'Leader2': 'M 5:00', 'Leader3': 'W 1
(Time: 1.1e-05 Seconds)
```

```
Leader0: T 7:00
Leader1: M 6:00
Leader2: M 5:00
Leader3: W 12:30
Leader4: F 5:00
Leader5: M 4:00
```

In this case, let us note that not all of the assignments are identical. This is due to the influence of the inference/heuristics on determining the solution. Evidently, the CSP is underconstrained, indicating that there are multiple correct solutions. Whereas the map-coloring problem output the same solution, the additional variation in this problem highlights the impact of the inference (`AC_3`) and heuristics (`MRV` , `LCV`).