# CISC 4/681 Programming Assignment 2

April 3, 2023

## Assignment Objectives

After completing this assignment, you'll have gained experience formulating and working with Constraint Satisfaction Problems. You'll understand the inner workings of the following algorithms:

- AC-3 and Maintaining Arc Consistency
- Backtracking Search for CSPs

You'll also gain some experience setting up a basic web service to visualize the results of your backtracking search.

## Sudoku

Sudoku is a number puzzle set on a $9 \times 9$ grid. The puzzle involves placing a digit in each cell - a solution having the property that each row, column, and each of the nine $3 \times 3$ non-overlapping subgrids (*boxes*) contains all of the digits $1$ thru $9$. A puzzle is defined by a partially filled out grid, and the solver must fill in the missing numbers. See Figure 1 for an example puzzle, and Figure 2 for its solution.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **7** | | | **4** | | | | **8** | **6** |
| | **5** | **1** | | **8** | | **4** | | |
| | **4** | | **3** | | **7** | | **9** | |
| **3** | | **9** | | | **6** | **1** | | |
| | | | | **2** | | | | |
| | | **4** | **9** | | | **7** | | **8** |
| | **8** | | **1** | | **2** | | **6** | |
| | | **6** | | **5** | | **9** | **1** | |
| **2** | **1** | | | | **3** | | | **5** |

**Figure 1:** An example Sudoku Puzzle. This is `*puzzle-1*` from the example puzzles.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **7** | 9 | 3 | **4** | 1 | 5 | 2 | **8** | **6** |
| 6 | **5** | **1** | 2 | **8** | 9 | **4** | 7 | 3 |
| 8 | **4** | 2 | **3** | 6 | **7** | 5 | **9** | 1 |
| **3** | 2 | **9** | 8 | 7 | **6** | **1** | 5 | 4 |
| 1 | 7 | 8 | 5 | **2** | 4 | 6 | 3 | 9 |
| 5 | 6 | **4** | **9** | 3 | 1 | **7** | 2 | **8** |
| 9 | **8** | 5 | **1** | 4 | **2** | 3 | **6** | 7 |
| 4 | 3 | **6** | 7 | **5** | 8 | **9** | **1** | 2 |
| **2** | **1** | 7 | 6 | 9 | **3** | 8 | 4 | **5** |

**Figure 2:** The solution to `*puzzle-1*`.

## Representing the Problem

*[My examples here are in Lisp, but feel free to use something similar that's easy to parse in your target language.]*

Recall that a Constraint Satisfaction Problem Consists of a set of *variables*, a set of *domains* (one for each variable), and a set of *constraints* over the variables.

One way to represent a Sudoku problem is as a *binary constraint satisfaction problem*. There will be $81$ variables in total - one for each cell on the board. There will be a constraint for each pair of variables in a row, column, or box specifying that these variables must have distinct values. We do this by specifying all distinct pairs $(i, j)$ such that both $i$ and $j$ are between $1$ and $9$ inclusive. Individual puzzles are specified by setting the domain for each variable - variables corresponding to blank squares will have the domain $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, and variables corresponding to squares with a number will have a domain containing just that number.

Imagine a *very* simplified version of the problem on a $2 \times 2$ grid, where a solution consists of making sure each row and column has the digits $1$ and $2$. Hopefully Figure 3, a representation of this very simple problem in Lisp as a CSP, helps illustrate how the representation of the full problem will look.

```
(:VARIABLES ((:C11 1) (:C12 1 2) (:C21 1 2) (:C22 1 2))
 :CONSTRAINTS (((:C11 :C12) (1 2) (2 1))
               ((:C11 :C21) (1 2) (2 1))
               ((:C21 :C22) (1 2) (2 1))
               ((:C12 :C22) (1 2) (2 1))))
```

**Figure 3:** A Lisp representation of the simplified $2 \times 2$ Sudoku as a CSP. In this puzzle, the upper-left square is set to $1$ and the other three squares are blank. **NB** that we've combined the variables and their domains into association lists, one for each variable. The constraints are lists of pairs, the first pair being the two variables involved in the constraint, and the remaining pairs comprising a complete list of compatible assignments for those two variables.

## Part 1 [5 pts]

You don't have to generate the constraints for the full $9 \times 9$ puzzle -

I've already generated those for you - but just to get the idea of how you specify Sudoku as a binary CSP, you should specify the $4 \times 4$ puzzle in Figure 4. The rules for this are the same as for the bigger game - each row, column, and box should contain the digits $1, 2, 3,$ and $4$.

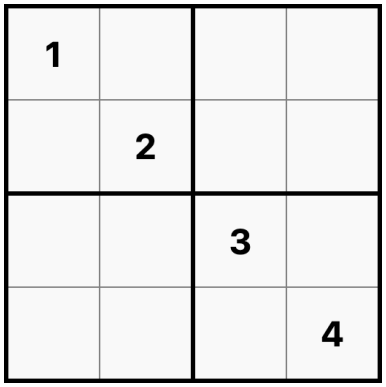


**Figure 4:** A simple $4 \times 4$ "mini Sudoku" puzzle.

## Part 2 [10 pts]

Write a function `revise` that takes a CSP such as the one in Figure 3 (or what you defined in Part 1) and the names of two variables as input and modifies the CSP, removing any value in the first variable's domain where there isn't a corresponding value in the other variable's domain that satisfies the constraint between the variables. The function should return a boolean indicating whether or not any values were removed.

## Part 3 [20 pts]

Implement AC-3 as a function which takes as input a CSP and modifies it such that any inconsistent values across all domains are removed. The function should return a boolean indicating whether or not all variables have at least on value left in their domains.

## Part 4 [5 pts]

Write a function `minimum-remaining-values` that takes a CSP and a set of variable assignments as input, and returns the variable with the fewest values in its domain among the unassigned variables in the CSP.

## Part 5 [35 pts]

Implement a *backtracking search* which takes a CSP and finds a valid assignment for all the variables in the CSP, if one exists. It should leverage your AC-3 implementation to maintain arc consistency.[1] When Choosing a variable to assign, it should use your minimum remaining values heuristic implementation. Along with the solution to the CSP, your search should return the order in which it assigned the variables, and the domains of the remaining unassigned variables after each assignment. You can test your code on the small problem you specified in Part 1, as well as the puzzles from the Example Puzzles section below.

[1] Technically, the MAC algorithm initializes the queue to only the edges involving the variable that's just been chosen for assignment - but doing a full run of AC-3 will achieve the same thing, at the cost of maybe doing a little bit of unnecessary work.

### Part 5 Fancy [481: 5 pts EC; 681: 10 of above 35 pts]

Your backtracking search should, for each variable, keep track of any *failed* values that it had to backtrack for. You'll probably also want to do some book keeping to keep track of, for each variable, how many previous variables required at least one backtrack to find the correct value.

## Part 6 [25 pts]

Now you'll implement a web-based visualization of your backtracking search.[2] It should allow the user to specify a Sudoku puzzle[3]. It should then show the board state at each step of the problem[4]. Figures 1 and 2 should give you an idea of how you might draw your board. You can use the web based version of the puzzles we examined in class to get started with the HTML/CSS for drawing your boards.

### Part 6 Fancy [481: 5 pts EC; 681: 5 of above 25 pts]

When displaying a board, change the color of the numbers being filled in going forward at every point where the algorithm had to make a guess when choosing a value for a cell. If the algorithm guessed wrong and had to backtrack, you should list the *failed* values in that cell, crossed out. See Figure 5 for an example of how this should look.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | 6 | 4 | 1 | **9** | 8 | 3 | **7** | **5** |
| 8 | 5 | **1** | **2** | 3 | 7 | 4 | 6 | 9, ~~6~~ |
| 9 | **7** | 3 | 4 | 6 | 5 | **1** | **8** | 2 |
| **3** | 2 | 7 | **6** | 4 | 1 | **9** | 5 | 8 |
| **1** | 8, ~~2~~ | 9 | 7 | **5** | 3 | 6 | 2 | **4** |
| 5 | 4 | **6** | 9 | 8 | **2** | 7 | 1 | **3** |
| 6 | **3** | **2** | 5 | 7 | 9 | 8 | **4** | 1 |
| 4 | 1 | 8 | 3 | 2 | **6** | **5** | 9 | 7 |
| **7** | **9** | 5 | 8 | **1** | 4 | 2 | 3 | 6 |

**Figure 5:** The solution to Puzzle 4, with colors and strikethrus indicating backtracking points.

[2] For Lisp, see Caveman 2 for a relatively easy to get up and running web framework.

[3] You don't have to get fancy, here - in my reference solution, I simply programmed it to accept as a query parameter a puzzle in the same format that the Example Puzzles are specified, flattened into one line.

[4] For a puzzle with $n$ blanks there will be $n$ steps.

## Example Puzzles

```
(defvar *puzzle-1*
  '((7 nil nil 4 nil nil nil 8 6)
    (nil 5 1 nil 8 nil 4 nil nil)
    (nil 4 nil 3 nil 7 nil 9 nil)
    (3 nil 9 nil nil 6 1 nil nil)
    (nil nil nil nil 2 nil nil nil nil)
    (nil nil 4 9 nil nil 7 nil 8)
    (nil 8 nil 1 nil 2 nil 6 nil)
    (nil nil 6 nil 5 nil 9 1 nil)
    (2 1 nil nil nil 3 nil nil 5)))

(defvar *puzzle-2*
  '((1 nil nil 2 nil 3 8 nil nil)
    (nil 8 2 nil 6 nil 1 nil nil)
    (7 nil nil nil nil 1 6 4 nil)
    (3 nil nil nil 9 5 nil 2 nil)
    (nil 7 nil nil nil nil nil 1 nil)
    (nil 9 nil 3 1 nil nil nil 6)
    (nil 5 3 6 nil nil nil nil 1)
    (nil nil 7 nil 2 nil 3 9 nil)
    (nil nil 4 1 nil 9 nil nil 5)))

(defvar *puzzle-3*
  '((1 nil nil 8 4 nil nil 5 nil)
    (5 nil nil 9 nil nil 8 nil 3)
    (7 nil nil nil 6 nil 1 nil nil)
    (nil 1 nil 5 nil 2 nil 3 nil)
    (nil 7 5 nil nil nil 2 6 nil)
    (nil 3 nil 6 nil 9 nil 4 nil)
    (nil nil 7 nil 5 nil nil nil 6)
    (4 nil 1 nil nil 6 nil nil 7)
    (nil 6 nil nil 9 4 nil nil 2)))
```

**Figure 6:** Lisp representations of several example Sudoku puzzles that you can use to test your algorithm.

```
(defvar *puzzle-4*
  '((nil nil nil nil 9 nil nil 7 5)
    (nil nil 1 2 nil nil nil nil nil)
    (nil 7 nil nil nil nil 1 8 nil)
    (3 nil nil 6 nil nil 9 nil nil)
    (1 nil nil nil 5 nil nil nil 4)
    (nil nil 6 nil nil 2 nil nil 3)
    (nil 3 2 nil nil nil nil 4 nil)
    (nil nil nil nil nil 6 5 nil nil)
    (7 9 nil nil 1 nil nil nil nil)))

(defvar *puzzle-5*
  '((nil nil nil nil nil 6 nil 8 nil)
    (3 nil nil nil nil 2 7 nil nil)
    (7 nil 5 1 nil nil 6 nil nil)
    (nil nil 9 4 nil nil nil nil nil)
    (nil 8 nil nil 9 nil nil 2 nil)
    (nil nil nil nil nil 8 3 nil nil)
    (nil nil 4 nil nil 7 8 nil 5)
    (nil nil 2 8 nil nil nil nil 6)
    (nil 5 nil 9 nil nil nil nil nil)))
```

**Figure 7:** Lisp representations of some more example Sudoku puzzles. These two puzzles are harder for a human to solve than `*puzzle-1*`, `*puzzle-2*`, and `*puzzle-3*` - but your program shouldn't have any trouble with them!

## Submitting

You should submit all of your code (including the CSP representation of the "mini Sudoku" in Part 1) - *document it appropriately, as you'll be graded on style.* You should also submit instructions for how to setup the environment for your web service and how to run it.[5,6]

[5] If you're using Lisp, you can just put any extra dependencies you added in the .asd file for the project.
[6] *e.g.* For Python this would include a requirements file that you would get from running `pip freeze`.