# CSE 368: Assignment 2

## Introduction

In this assignment you will use both the AC3 algorithm and Backtracking search to solve a Sudoku problem. There are 3 functions you need to edit, all of which are contained in the solver.py file. The other file, sudoko.py contains helper functions and classes that will aid you in finding the solution to this CSP.

## Solver.py

Recall that in a Constraint Satisfaction Problem we have the following:

- Set of variables X: {X1 , X2 , … , Xn }
- Set of Domains D: {D1 , D2 , … , Dn }
- Set of Constraints C: {C1 , C2 , … , Cn }
    - Each constraint, Ci , is a pair (scope, rel) • scope – tuple of variables that participate in constraint • rel – relationship that defines values variables can have • Ex. C1 = ( (X1 , X2 ), X1 > X2 )

Class **CSP()** essentially defines the above, but with a few more helpful methods. Here are some useful variables from this class:

- variables is a list containing all the variables on the sudoku board (goes from 0 to 81 on a 9x9 grid)
- domains is a dictionary containing the domains of all the variables on the board. The keys are the variables, and the values are the set of all possible values it can take
    - Ex. {0: ['1', '2', '3', '4', '5', '6', '7', '8', '9'], …}
- curr_domains is a dictionary containing the domains of all the variables on the board. The keys are the variables, and the values are the set of all possible values it can take
    - Ex. {0: ['1', '2', '3', '4', '5', '6', '7', '8', '9'], …}
    - The main difference between this and the domain variable is that when you call the *prune* method to remove values from the domain, curr_domains will be changed instead of domain
- neighbors is a dictionary of sets, which contains the neighbors for a given variable. In the context of Sudoku, the neighbors will be everything in the same grid, the same row, and column. For example, neighbors[0] returns a set (a collection of unordered, unindexed, unique items) which is {1, 2, 3, 4, 5, 6, 7, 8,

9, 10, 11, 18, 19, 20, 27, 30, 33, 54, 57, 60} (we will talk about why it's like this soon)

Here are some useful methods from CSP:

- *prune(self, var, value, removals)*: This method removes value from the domain of the variable, var. Removals will be a list of tuples containing everything you've removed so far. I.e. [ (3, '8'), (23, '1'), (23, '2'), (23, '3') ]

- *constraints(A, a, B, b)*: Function that returns true if neighbors A and B satisfy the constraint, which is that their values a and b are not the same

- *nconflicts(self, var, val, assignment)*: This function returns the number of conflicts that the variable var has when it's value is set to val

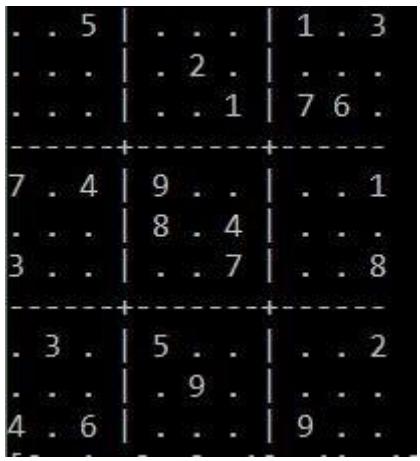
- *suppose(self, var, value)*: Returns a list of tuples that contains the values that var can no longer take. For example, suppose tile 0 can be any value from 0 - 9, calling csp.suppose(0, '3') returns [(0, '1'), (0, '2'), (0, '4'), (0, '5'), (0, '6'), (0, '7'), (0, '8'), (0, '9')]

- *restore(self, removals)*: This function will undo everything that suppose did. For example, if we called csp.restore( [(0, '1'), (0, '2'), (0, '4'), (0, '5'), (0, '6'), (0, '7'), (0, '8'), (0, '9'), (3, '2')] this will add back the values 1,2,4,5,6,7,8,9 to the domain of 0 and 2 to the domain of 3

- *unassign(self, var, assignment)*: assignment will be a dictionary containing the assigned values for a given variable. This function removes var from that dictionary

- *assign(self, var, assignment)*:This function adds a variable var, to the dictionary assignment

The only useful method in **Sudoku()** is the display method, which will display an image that looks like so:




The indices for each of the positions in the board would look like this:

0 1 2 | 9 10 11 | 18 19 20 | 3
4 5 | 12 13 14| 21 22 23 |
6 7 8 | 15 16 17| 24 25 26 |
--------|------------|--------------|
27 28 29 | 36 37 38| 45 46 47 |
30 31 32 | 39 40 41| 48 49 50 |
33 34 35 | 42 43 44| 51 52 53 |
------------|-------------|---------------|
54 55 56 | 63 64 65 | 72 73 74 |
57 58 59 | 66 67 68 | 75 76 77 |
60 61 62 | 69 70 71 | 78 79 80 |
------------|------------|---------------|

# Written Report

**(Required)** : For each of the algorithms you will be implementing (listed below), please provide a written report that includes a description of the algorithm, the results you got from running the algorithm and how well your implementation is working **even if you do not finish.**

**(50 points) AC3 Algorithm:** For implementing the AC3 algorithm, you will be editing the method AC3(self, csp, queue=None, removals=None) as well as its helper function that you will use, revise(self, csp, $X_i$, $X_j$, removals). This algorithm will take a

csp as input, and reduce its domains. It should return True if there are any possible assignments left, and False if the algorithm concludes the CSP can't be solved.

- ❏ The revise() function should check if the neighbors $X_i$ and $X_j$ are arc consistent. If they are, return True and if they aren't then return False.

**(50 points) Backtracking Search:** Implement a backtracking search for solving a Sudoku puzzle. The function backtracking_search(self, csp) is a recursive function which only calls on backtrack(self, assignment, csp). This is where you will implement the backtrack algorithm; it either returns a valid assignment for each variable, or None if there is none.

- *Hint*: You will know when there is a valid assignment when the length of the assignment variable is 81. This means that each variable (0-80) have a valid assignment
- Three more functions that will be of use for backtrack are:
  - *order_domain_values(self, var, assignment, csp):* This function returns a list of domain values for a given variable, that are sorted by how many conflicts they have.
  - *select_unassigned_variable(self, assignment, csp):* This function returns a variable which hasn't been assigned a value.
  - *get_queue(self, csp, var):* This function returns a list of tuples, where the first element is the neighbor of var, and the second is var.
- Lastly, make sure you are calling AC3 inside of this function to maintain arc consistency.

**(Extra Credit: 20 points) Solve 12 x 12 Sudoku:** Implement both AC3 and backtracking search to solve 12 x 12 sudoku. Your runtime matters here, and will be more strictly graded. Separately submit this file on Autograder. It will not be autograded.

# Submission Details

Please submit all code to auto grader for grading purposes. Condense all of the written reports into a **single PDF file**. Then submit a **zip file** containing **both** your code and written report to UBLearns.