CP468 Assignment 2 – CSP Sudoku Solver

Group 8

Dylan Clarry – 160718680
Aayush Sheth – 160642890
Adam Gumieniak – 160446280
Bryan Mietkiewicz – 160713690

Monday, November 11, 2019

Variables, Domains, Constraints

Our approach to the CSP Sudoku solver was accomplished by using relevant data structures which compliment our problem. We used a list to hold our variables for our sudoku board; each index represents a cell on the board. We labelled the rows alphabetically and the columns numerically where the first grid cell is value A1 and the last one is I8. To store the row, column & grid values we used a Python dictionary, this information is relevant when applying arc consistency. For our domains we also used a dictionary variable where we used the cell as the key to retrieve the corresponding domain as the value. We created binary constraints from the 27-alldiff constraints when revising during AC3.

Variables: [A1, A2, A3, …, B1, B2, B3, …, I8] A list is used to represent all of the cell variables in the sudoku puzzle

Domains: { (1-81) : [1-9] } Each cell in the sudoku puzzle from A1 to I8 will have a domain with values from 1 to 9 that are then reduced down by our AC-3 algorithm

Constraints: When we revise our domains during AC3, we test two chosen cells Xi and Xj who are neighbors and compare their domains against each other. If they happen to have matching values, then this violates our constraints as no two neighbors can have the same value. We then delete the matching value from the domain of Xi as per the rules of AC3. By comparing these two variables we are treating them as our binary constraints. This is the method we have chosen to check our constraints, however, if we were to revisit this component of the sudoku solver we would implement a separate data structure to store the constraints by row, column, and grid. Then, we would work through the constraints to permutate the constraints down to 2 which would make working with binary constraints much more efficient.

Input & Output

For our input method we decided to have the user input a one-line sudoku puzzle into the terminal; all 81 sudoku puzzle cells are represented in one line.

Ex. 003020600900305001001806400008102900700000008006708200002609500800203009005010300

To display the output of the solved sudoku puzzle we print it with column numbers on top and row letters on the left side. If the user inputs an unsolvable sudoku puzzle nothing will print out and tell the user that the given sudoku puzzle has no solution.

Program

```
import random
import sys

# prints the board in a readable format
def print_board(board):
    row_labels = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']
    print ("\n   1 2 3 4 5 6 7 8 9\n")
    for x in range(0,9):
        word = row_labels[x] + '  '
        for y in range(0, 9):
            if(str(board[9 * x + y]) == '0'):
                word += '- '
            else:
                word += str(board[9 * x + y]) + ' '
        print(word)

# checks if the board is solved
def solved(domains, grids, variables):
    for e in variables:
        for neigh in grids:
            answers = list(range(1,10))
            for xi in neigh:
                if(int(domains[xi][0]) in answers):
                    answers.remove(int(domains[xi][0]))
                else:
                    return False
    return True

# checks to make sure the number is valid
def valid_choice(xi, constraints, assignment):

    for neigh in constraints[xi]:
        answers = list(range(1,10))
        if(int(assignment[xi][0]) != 0):
            answers.remove(int(assignment[xi][0]))

        for value in neigh:
            complete = False
            for p_val in assignment[value]:

                if(complete == False and int(p_val) != 0 and p_val in 
answers):
                    answers.remove(int(p_val))
                    complete = True

                if(complete == False and int(p_val) == 0):
                    complete = True

            if(complete == False):
                return False

    return True
```

```
# recursing backtracking algorithm
def backtracking(val, assignment, variables, constraints, orig_domains,
grids):
    if(solved(assignment,grids,variables)):
        return True

    for x in orig_domains[val]:
        assignment[val] = [x]
        if(valid_choice(val, constraints, assignment)):

            passed = 0
            next_val = 0
            for var in variables:
                if(passed == 1 and len(orig_domains[var]) > 1):
                    next_val = var
                    break
                if(var == val):
                    passed = 1

            if(next_val != 0):
                if(backtracking(next_val, assignment, variables, constraints,
orig_domains, grids)):
                    return True
            else:
                if(solved(assignment, grids, variables)):
                    return True

    assignment[val] = [0]
    return False

# initialize the arcs
def initialize_arcs(variables, constraints):
    arcs = []
    for xi in variables:
        for neigh in constraints[xi]:
            for xj in neigh:
                arcs.append((xi,xj))
    return arcs

# create sudoku cell variables
def create_variables(column_letters):
    variables = []

    for letters in column_letters:
        for numbers in range(0, 9):
            variables.append(str(letters) + str(numbers))
    return variables

# creates the grids
def create_grids():
    return [
        ['A0', 'A1', 'A2', 'B0', 'B1', 'B2', 'C0', 'C1', 'C2'],
        ['A3', 'A4', 'A5', 'B3', 'B4', 'B5', 'C3', 'C4', 'C5'],
        ['A6', 'A7', 'A8', 'B6', 'B7', 'B8', 'C6', 'C7', 'C8'],
        ['D0', 'D1', 'D2', 'E0', 'E1', 'E2', 'F0', 'F1', 'F2'],
        ['D3', 'D4', 'D5', 'E3', 'E4', 'E5', 'F3', 'F4', 'F5'],
        ['D6', 'D7', 'D8', 'E6', 'E7', 'E8', 'F6', 'F7', 'F8'],
```

```
        ['G0', 'G1', 'G2', 'H0', 'H1', 'H2', 'I0', 'I1', 'I2'],
        ['G3', 'G4', 'G5', 'H3', 'H4', 'H5', 'I3', 'I4', 'I5'],
        ['G6', 'G7', 'G8', 'H6', 'H7', 'H8', 'I6', 'I7', 'I8']
    ]

# create the constraints for each cell
def create_constraints(grids, variables, column_letters):
    constraints = {}

    for letters in column_letters:
        row = []
        for numbers in range(0,9):
            row.append(str(letters) + str(numbers))
        grids.append(row)

    # add columns to grids
    for numbers in range(0,9):
        row = []
        for letters in ['A','B','C','D','E','F','G','H','I']:
            row.append(str(letters) + str(numbers))
        grids.append(row)

    # add empty lists for our variables
    for xi in variables:
        constraints[xi] = []

    # filling in the constraints
    for rows in grids:
        for xi in rows:
            index = rows.index(xi)
            constraints[xi].append(rows[:index] + rows[index + 1:])

    return constraints

# create cell domains
def create_domains(variables, sudoku_input):
    domains = {}
    filled_cells = 0
    for index, xi in enumerate(variables):
        if(int(sudoku_input[index]) == 0):
            domains[xi] = [1,2,3,4,5,6,7,8,9]
        else:
            domains[xi] = [int(sudoku_input[index])]
            filled_cells += 1
    return domains, filled_cells

# main function
def main():
    sudoku_input = input('Input an unsolved 9x9 unsolved sudoku: \n')
    sudoku_input = sudoku_input[:81]
    print("\nInput Board:\n")
    print_board(sudoku_input)

    # initialize variables
    column_letters = ['A','B','C','D','E','F','G','H','I']
    variables = create_variables(column_letters)
```

```
    # get our grid constraints
    grids = create_grids()
    constraints = create_constraints(grids, variables, column_letters)

    # create domains
    domains, filled_cells = create_domains(variables, sudoku_input)

    # any sudoku with less than 17 clues is not solvable
    if filled_cells < 17:
        print ("\nError! This board is not Solvable!!")
        return

    arcs = initialize_arcs(variables, constraints)

    Solution = True
    ac3_xis = 0

    # ac3 implementation
    while arcs:
        xi, xj = arcs.pop(0)

        # Revised Func
        revised = False
        for xi_sol in domains[xi]:
            passed = False
            for xj_sol in domains[xj]:
                if(xi_sol != xj_sol):
                    passed = True

            if(passed == False):
                #print("Domain Before Revision:", domains[xi])
                domains[xi].remove(xi_sol)
                #print("Domain Before Revision:", domains[xi])
                revised = True
                ac3_xis += 1

        if revised:
            if (len(domains[xi]) == 0):
                Solution = False
            for neigh in constraints[xi]:
                for xi3 in neigh:
                    if(xi3 != xj):
                        arcs.append((xi3, xi))

    selected = []
    answer_dict = dict()

    next_val = 0
    assignment = dict()

    # assign values
    for var in variables:
        if(len(domains[var]) == 1):
            assignment[var] = domains[var]
        else:
            assignment[var] = [0]
```

```
    for var in variables:
        if(len(domains[var]) > 1):
            next_val = var
            break

    if(next_val != 0):
        backtracking(next_val, assignment, variables, constraints, domains,
grids)
        answer_dict = assignment
    else:
        answer_dict = domains

    answer = ''
    solution_found = True

    for index, xi in enumerate(variables):
        try:
            answer = answer + str(answer_dict[xi][0])
        except IndexError:
            solution_found = False

    if(solution_found):
        print("\n")
        print ("Output Board:")
        print_board(answer)
    else:
        print ("Err: Board not solvable")
    return

main()
```

Input Puzzle & Result

Revision Output:

```
Domain Before Revision: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Domain Before Revision: [1, 2, 3, 4, 5, 7, 8, 9]
Domain Before Revision: [1, 2, 3, 4, 5, 7, 8, 9]
Domain Before Revision: [1, 2, 3, 4, 5, 7, 8]
Domain Before Revision: [1, 2, 3, 4, 5, 7, 8]
Domain Before Revision: [1, 3, 4, 5, 7, 8]
Domain Before Revision: [1, 3, 4, 5, 7, 8]
Domain Before Revision: [1, 4, 5, 7, 8]
Domain Before Revision: [1, 4, 5, 7, 8]
Domain Before Revision: [1, 5, 7, 8]
Domain Before Revision: [1, 5, 7, 8]
Domain Before Revision: [5, 7, 8]
Domain Before Revision: [5, 7, 8]
Domain Before Revision: [7, 8]
Domain Before Revision: [7, 8]
Domain Before Revision: [7]
Domain Before Revision: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Domain Before Revision: [1, 2, 3, 4, 6, 7, 8, 9]
Domain Before Revision: [1, 2, 3, 4, 6, 7, 8, 9]
Domain Before Revision: [1, 2, 3, 4, 6, 8, 9]
Domain Before Revision: [1, 2, 3, 4, 6, 8, 9]
Domain Before Revision: [1, 2, 3, 4, 6, 8]
Domain Before Revision: [1, 2, 3, 4, 6, 8]
Domain Before Revision: [1, 2, 4, 6, 8]
Domain Before Revision: [1, 2, 4, 6, 8]
Domain Before Revision: [1, 2, 6, 8]
Domain Before Revision: [1, 2, 6, 8]
Domain Before Revision: [2, 6, 8]
Domain Before Revision: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Domain Before Revision: [1, 2, 3, 4, 6, 7, 8, 9]
Domain Before Revision: [1, 2, 3, 4, 6, 7, 8, 9]
Domain Before Revision: [1, 2, 3, 4, 6, 8, 9]
Domain Before Revision: [1, 2, 3, 4, 6, 8, 9]
Domain Before Revision: [1, 2, 3, 4, 6, 8]
Domain Before Revision: [1, 2, 3, 4, 6, 8]
Domain Before Revision: [1, 2, 4, 6, 8]
Domain Before Revision: [1, 2, 4, 6, 8]
Domain Before Revision: [1, 2, 6, 8]
Domain Before Revision: [1, 2, 6, 8]
Domain Before Revision: [2, 6, 8]
Domain Before Revision: [2, 6, 8]
Domain Before Revision: [2, 6]
Domain Before Revision: [3, 4, 5, 6]
Domain Before Revision: [3, 5, 6]
Domain Before Revision: [4, 5, 9]
Domain Before Revision: [5, 9]
Domain Before Revision: [3, 4, 5, 6, 9]
Domain Before Revision: [3, 5, 6, 9]
Domain Before Revision: [1, 2, 3, 4, 5, 9]
Domain Before Revision: [1, 2, 3, 5, 9]
Domain Before Revision: [4, 9]
Domain Before Revision: [9]
Domain Before Revision: [1, 4, 7]
Domain Before Revision: [1, 7]
Domain Before Revision: [1, 2, 3, 5, 9]
Domain Before Revision: [2, 3, 5, 9]
Domain Before Revision: [1, 4, 7, 8]
Domain Before Revision: [1, 4, 8]
Domain Before Revision: [4, 7]
Domain Before Revision: [4]
Domain Before Revision: [1, 4, 6, 7]
Domain Before Revision: [1, 4, 6]
```

Puzzle 1:

```
[dylanclarry-> python3 AC3.py
 Input an unsolved 9x9 unsolved sudoku:
 003020600900305001001806400008102900700000008006708200002609500800203009005010300

 Input Board:


    1 2 3 4 5 6 7 8 9

 A  - - 3 - 2 - 6 - -
 B  9 - - 3 - 5 - - 1
 C  - - 1 8 - 6 4 - -
 D  - - 8 1 - 2 9 - -
 E  7 - - - - - - - 8
 F  - - 6 7 - 8 2 - -
 G  - - 2 6 - 9 5 - -
 H  8 - - 2 - 3 - - 9
 I  - - 5 - 1 - 3 - -


 Output Board:

    1 2 3 4 5 6 7 8 9

 A  4 8 3 9 2 1 6 5 7
 B  9 6 7 3 4 5 8 2 1
 C  2 5 1 8 7 6 4 9 3
 D  5 4 8 1 3 2 9 7 6
 E  7 2 9 5 6 4 1 3 8
 F  1 3 6 7 9 8 2 4 5
 G  3 7 2 6 8 9 5 1 4
 H  8 1 4 2 5 3 7 6 9
 I  6 9 5 4 1 7 3 8 2
 ^_^[~/Documents/coursework/final year/CP468/Assignment_02/AI-Projects]
 dylanclarry->
```

Puzzle 2:

```
[dylanclarry-> python3 AC3.py
Input an unsolved 9x9 unsolved sudoku:
000008020000006930098070001000000000009210000700000096240090000000300180000000003

Input Board:


   1 2 3 4 5 6 7 8 9

A  - - - - - 8 - 2 -
B  - - - - - 6 9 3 -
C  - 9 8 - 7 - - - 1
D  - - - - - - - - -
E  - - 9 2 1 - - - -
F  7 - - - - - - 9 6
G  2 4 - - 9 - - - -
H  - - - 3 - - 1 8 -
I  - - - - - - - - 3


Output Board:

   1 2 3 4 5 6 7 8 9

A  1 5 6 9 3 8 4 2 7
B  4 2 7 1 5 6 9 3 8
C  3 9 8 4 7 2 5 6 1
D  5 3 4 6 8 9 7 1 2
E  8 6 9 2 1 7 3 5 4
F  7 1 2 5 4 3 8 9 6
G  2 4 3 8 9 1 6 7 5
H  6 7 5 3 2 4 1 8 9
I  9 8 1 7 6 5 2 4 3
^_^[~/Documents/coursework/final year/CP468/Assignment_02/AI-Projects]
dylanclarry->
```

Puzzle 3:

```
[dylanclarry-> python3 AC3.py
 Input an unsolved 9x9 unsolved sudoku:
 000000000000000000000000000000000000000000000000000000000000000000000000000001000

 Input Board:


    1 2 3 4 5 6 7 8 9

 A  - - - - - - - - - -
 B  - - - - - - - - - -
 C  - - - - - - - - - -
 D  - - - - - - - - - -
 E  - - - - - - - - - -
 F  - - - - - - - - - -
 G  - - - - - - - - - -
 H  - - - - - - - - - -
 I  - - - - - - 1 - - -

 Error! This board is not Solvable!!
 ^_^[~/Documents/coursework/final year/CP468/Assignment_02/AI-Projects]
 dylanclarry->
```