CP468 Assignment 2 – Sudoku as a CSP

Group ID 11

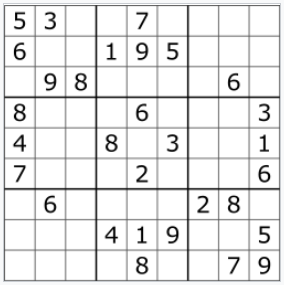
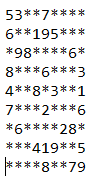
Lucas Boulanger, Vik Sharma, Karman Brar, Tomi Solaja

Sudoku

**Problem Definition**

A sudoku puzzle consists of a 9x9 grid. The goal of the sudoku is to fill the grid such that each column, each row, and each of the 9 3x3 sub grids (that the entire 9x9 grid is comprised of) contain all the digits from 1 to 9. The puzzle is given as a partially solved grid, with the objective to fill the remaining spaces. For this assignment, a sudoku is read from a file as a 9x9 grid of characters. If the character is a ‘\*’, then there is no partial answer to that grid location. Likewise, If a number is given, then that is the partial solution for that grid location.

**For example**: the following is a sample sudoku puzzle and its file input representation:



**As a CSP:**

A CSP is defined as a series of variables, as well as domains and constraints for those variables. In this sense, a sudoku can be represented as 81 coordinates (x, y) to represent each individual square.

For this project, the domains the CSP is represented by a single python dictionary, where the keys are each coordinate and the value is the domain for that coordinate. The domain is represented as a list of integers containing the possible values that can be assigned to that coordinate. If the domain is size 1, then that coordinate has been assigned a value.

Constraints are similarly represented as a python dictionary with the keys being the 81 coordinates. Each value is a list of coordinates that the key is a neighbor two. This is the list of coordinates the key shares a row with, a column with, and a box with. These constraints are checked by checking values in the domain in the key to values in the domains of the neighbors to check if they are equal (If values are equal, then they are not valid).

The program performs AC3 on the sudoku to maintain arc-consistency. If the sudoku is deemed to be arc-consistent, but no solution is immediately found, backtracking with MCV is performed. Arc consistency is used in backtracking to determine solution.

**Code:**

**from** collections **import** deque

**import** copy

**import** sys

**import** time

**class** **sudokuCSP():**

**def** \_\_init\_\_**(**self**,** file**):**

'''

Initializes a sudoku CSP problem.

Domain is a dictionary where keys are coordinates (x, y) and values are the domain.

- Default domain values are [1,2,3,4,5,6,7,8,9] for empty squares, and [i] for non-empty squares assigned value i.

- Sudoku Variable values are inferred from the domain.

If length(domain[key]) > 1, then value at key is unassigned.

if length(domain[key]) = 1, then the value at key is assigned domain[key][0].

if length(domain[key]) = 0, then no value is possible.

constraints is a dictionary where keys are coordinates (x, y) and values are that coordinates neighbors.

- coordinates are neighbors to other coordinates that share a row, column, and/or 3x3 box.

constraints are checked by looking at values at coordinates (in domain) in constraints

'''

self**.**domain **=** **{}**

self**.**constraints **=** **{}**

x **=** 0

y **=** 0

#read puzzle from file

**for** line **in** file**:**

**for** char **in** line**:**

**if** char **!=** "\n"**:**

self**.**constraints**[(**x**,**y**)]** **=** **[]**

**if** char **==** "\*"**:**

#if no value give, domain becomes 1 to 9

self**.**domain**[(**x**,**y**)]** **=** **[**1**,**2**,**3**,**4**,**5**,**6**,**7**,**8**,**9**]**

**else:**

#otherwise, domain is limited to the value given

self**.**domain**[(**x**,**y**)]** **=** **[**int**(**char**)]**

y **+=** 1

x **+=** 1

y **=** 0

#adds all possible constrait arcs

**for** variable **in** self**.**constraints**:**

#go through all 81 variables

**for** i **in** range **(**0**,**9**):**

**for** j **in** range **(**0**,**9**):**

#gathers all points the current 'variable' is a neighbor to (row, column, and box for the three or parts respectively)

**if** **(**variable**[**0**]** **==** i **and** variable**[**1**]** **!=** j**)** **or** **(**variable**[**0**]** **!=** i **and** variable**[**1**]** **==** j**)** **or** **(**variable**[**0**]** **in** range**((**i**//**3**)\***3**,** **(**i**//**3**)\***3 **+** 3**)** **and** variable**[**1**]** **in** range**((**j**//**3**)\***3**,** **(**j**//**3**)\***3 **+** 3**)):**

#add point to constraints as an arc from key to value

**if** **not(**variable**[**0**]** **==** i **and** variable**[**1**]** **==** j**):**

self**.**constraints**[**variable**].**append**((**i**,**j**))**

**return**

# check two values. If two values are equal, then they violate the binary isDifferent contraint between the

# coordinate pairs the values came from.

**def** constraintCheck**(**self**,** x**,** y**):**

**return** x **!=** y

# prints domain.

**def** printDomain**(**self**):**

**for** key **in** self**.**domain**:**

**print(**"{} -> {}"**.**format**(**key**,** self**.**domain**[**key**]))**

**return**

#determines if current CSP is solved

**def** isSolved**(**self**):**

**for** key **in** self**.**domain**:**

# only solved if length of every domain is 1

**if** len**(**self**.**domain**[**key**])** **!=** 1**:**

**return** **False**

**return** **True**

#prints the puzzle nice and pretty like

**def** \_\_str\_\_**(**self**):**

s **=** ""

count **=** 0

rows **=** 0

**for** key **in** self**.**domain**:**

**if** **(**rows **==** 0 **or** rows **==** 3 **or** rows **==** 6**)** **and** count **==** 0**:**

s **+=** "----------------------\n"

**if** count **==** 0 **or** count **==** 3 **or** count **==** 6**:**

s **+=** "|"

**if** len**(**self**.**domain**[**key**])** **==** 1**:**

s **+=** str**(**self**.**domain**[**key**][**0**])** **+** " "

**else:**

s **+=** "\* "

count **+=** 1

**if** count **==** 9**:**

s **+=** "|\n"

rows **+=** 1

count **=** 0

**return** s **+** "----------------------"

**def** AC3**(**csp**):**

#initiate a queue to store all current constraint arcs as two sets of coordinates ((x1, x2), (y1, y2))

queue **=** deque**()**

**for** variable **in** csp**.**constraints**:**

**for** neighbor **in** csp**.**constraints**[**variable**]:**

queue**.**append**((**variable**,** neighbor**))**

#iterate through queue until it isnt empty

**while** queue**:**

#pop the next arc from the queue

Xi**,**Xj **=** queue**.**popleft**()**

#check for revised domains

**if** revise**(**csp**,** Xi**,**Xj**):**

#if the domain has become 0, then no solution is possible without violating constraints

**if** len**(**csp**.**domain**[**Xi**])** **==** 0**:**

**return** **False**

#add new arcs to queue as the domain of Xi has been updated

**for** variable **in** csp**.**constraints**[**Xi**]:**

# but don't include the arc for (Xi,Xj) as that is the arc we just accounted for

**if** **not** **(**variable**[**0**]** **==** Xj**[**0**]** **and** variable**[**1**]** **==** Xj**[**1**]):**

queue**.**append**((**variable**,** Xi**))**

#if the queue has been emptied, the algorithm has succeeded

**return** **True**

**def** revise**(**csp**,** Xi**,** Xj**):**

revised **=** **False**

#go through all x values in domain of Xi

**for** x **in** csp**.**domain**[**Xi**][:]:**

#if x does not satisfy the constraint for any possible y in the domain of Xj, then x must be removed

**if** **not** any**([**csp**.**constraintCheck**(**x**,** y**)** **for** y **in** csp**.**domain**[**Xj**]]):**

csp**.**domain**[**Xi**].**remove**(**x**)**

revised **=** **True**

#indicates if a value has been removed from the domain or not

**return** revised

created **=** 0

**def** backTrackingSearch**(**csp**):**

# empty is a list of tuples where a given value = ((x,y), [domain of (x,y)])

# representing all unanswered variables

DFS **=** deque**()**

DFS**.**append**(**copy**.**deepcopy**(**csp**))**

#number of expanded nodes we look at

**global** created

#while the queue is not empty, there are possible solutions that need to be analyzed

**while** DFS**:**

state **=** DFS**.**pop**()**

#check if the current state is arc-consistent, if not, it is an invalid state and should be discarded

**if** AC3**(**state**):**

#if the state is solved, then the puzzle has been completed

**if** state**.**isSolved**():** **return** state

#otherwise, grab the next unassigned variable to expand (as (M)ost (C)onstrained (V)ariable)

mostConstricted **=** **None**

domainsize **=** 10

#grab the variable with the smallest domain size:

**for** variable **in** state**.**domain**:**

length **=** len**(**state**.**domain**[**variable**])**

**if** domainsize **>** length **>** 1**:**

mostConstricted **=** variable

domainsize **=** length

#for x in state.domain[mostConstricted]:

**for** x **in** orderDomainValues**(**state**,** mostConstricted**):**

successorState **=** copy**.**deepcopy**(**state**)**

successorState**.**domain**[**mostConstricted**]** **=** **[**x**]**

created **+=** 1

DFS**.**append**(**successorState**)**

**return** **None**

#Didn't implement LCV,

**def** orderDomainValues**(**csp**,** variable**):**

unordered **=** csp**.**domain**[**variable**]**

**return** unordered

#Main Code

sudoku **=** sudokuCSP**(**open**(**"data/2012.txt"**,**"r"**))**

**print(**"\n\n\nCurrent sudoku puzzle: "**)**

**print(**sudoku**)**

**print(**"\nAttempting AC-3 algorithm: "**)**

**if** **not** AC3**(**sudoku**):**

**print(**"\nPuzzle not arc-consistent. Cannot solve, exiting."**)**

exit**()**

**if** sudoku**.**isSolved**():**

**print(**"\nSudoku solved by AC-3."**)**

**print(**sudoku**)**

**print(**"\n"**)**

exit**()**

**print(**"\nSudoku not solved by AC-3..."**)**

**print(**"\nEquivelent arc-consistent sudoku puzzle:\n"**,** sudoku**,** "\nWith domain set: "**)**

sudoku**.**printDomain**()**

**print(**"\nAttempting backtracking algorithm:"**)**

t1 **=** time**.**time**()**

**print(**backTrackingSearch**(**sudoku**))**

t2 **=** time**.**time**()**

**print** **(**"nodes created: "**,** created**)**

**if** **(**t2 **-** t1 **>=** 300**):**

**print(**"Yikes... "**,** end**=**""**)**

**print(**"Backtracking search completed in: {} seconds."**.**format**(**t2**-**t1**))**