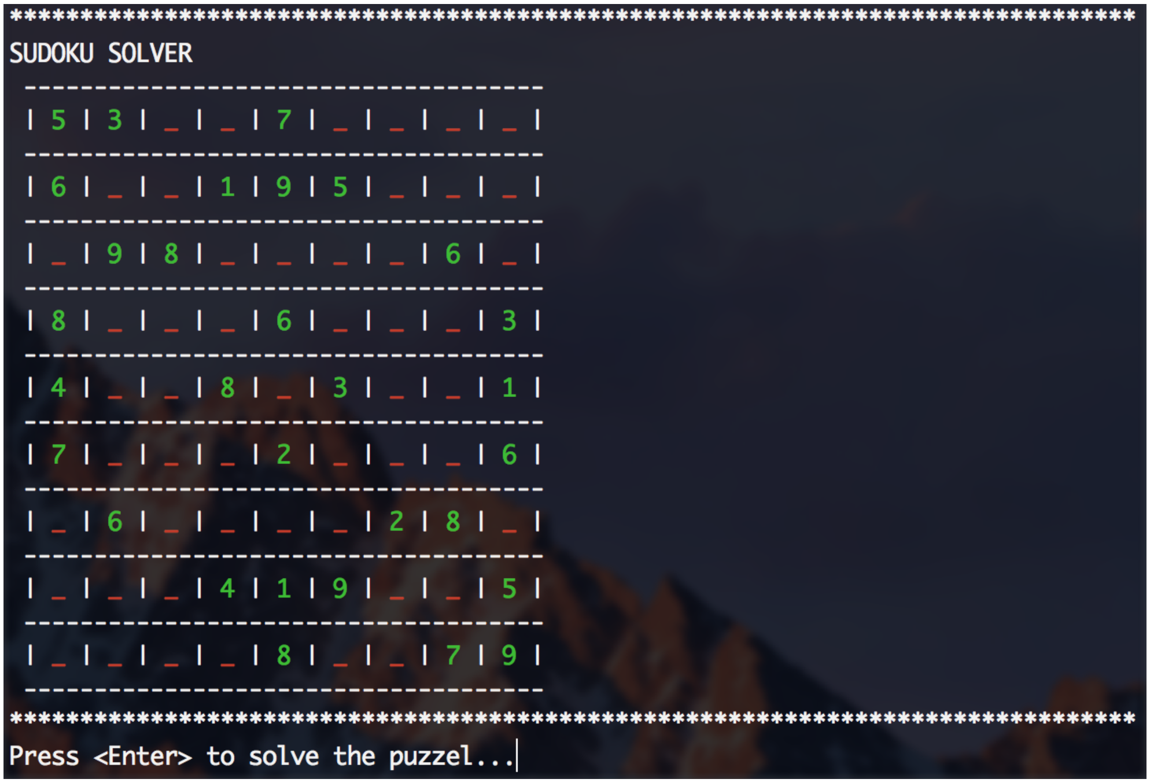
CP 468 Assignment 1
sudoku as a constraint satisfaction problem

Morouney, Robert
`robert@morouney.com, 069001422`

November 09 2016

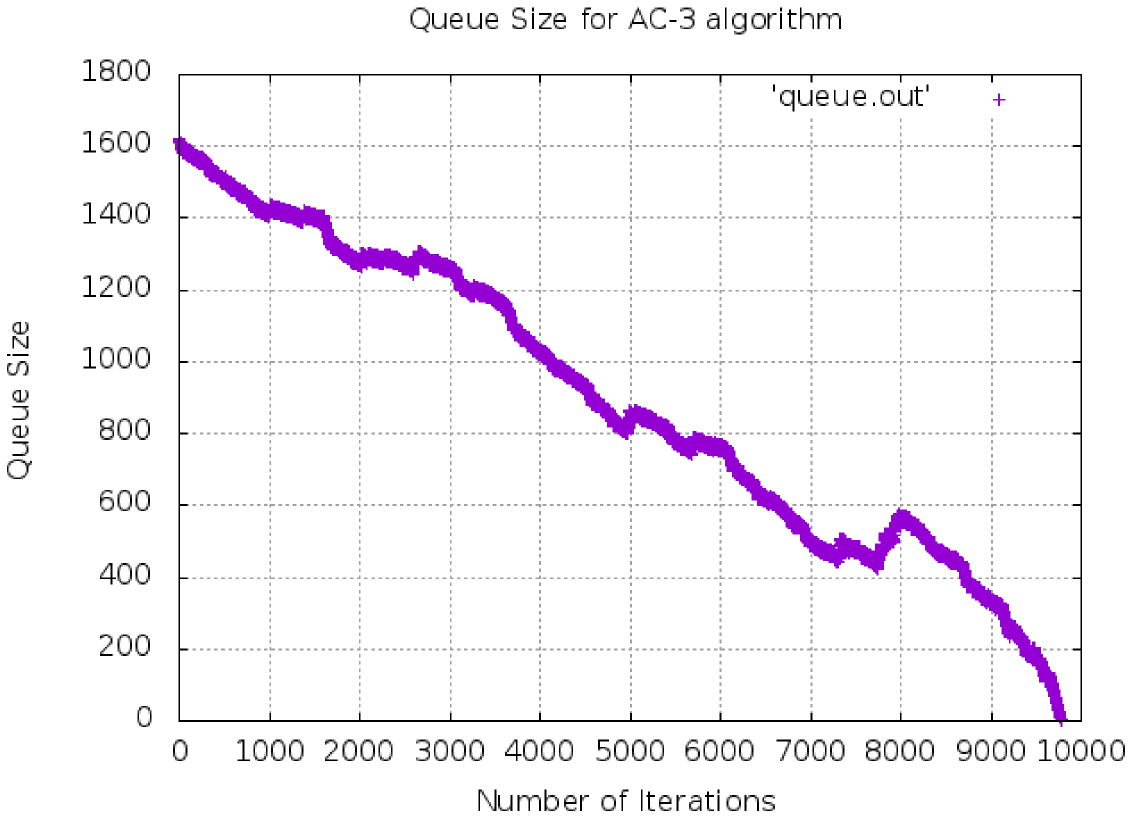
1
Sudoku



```
********************************************************************************
SUDOKU SOLVER
 -------------------------------------
 | 5 | 3 | _ | _ | 7 | _ | _ | _ | _ |
 -------------------------------------
 | 6 | _ | _ | 1 | 9 | 5 | _ | _ | _ |
 -------------------------------------
 | _ | 9 | 8 | _ | _ | _ | _ | 6 | _ |
 -------------------------------------
 | 8 | _ | _ | _ | 6 | _ | _ | _ | 3 |
 -------------------------------------
 | 4 | _ | _ | 8 | _ | 3 | _ | _ | 1 |
 -------------------------------------
 | 7 | _ | _ | _ | 2 | _ | _ | _ | 6 |
 -------------------------------------
 | _ | 6 | _ | _ | _ | _ | 2 | 8 | _ |
 -------------------------------------
 | _ | _ | _ | 4 | 1 | 9 | _ | _ | 5 |
 -------------------------------------
 | _ | _ | _ | _ | 8 | _ | _ | 7 | 9 |
 -------------------------------------
********************************************************************************
Press <Enter> to solve the puzzel...
```

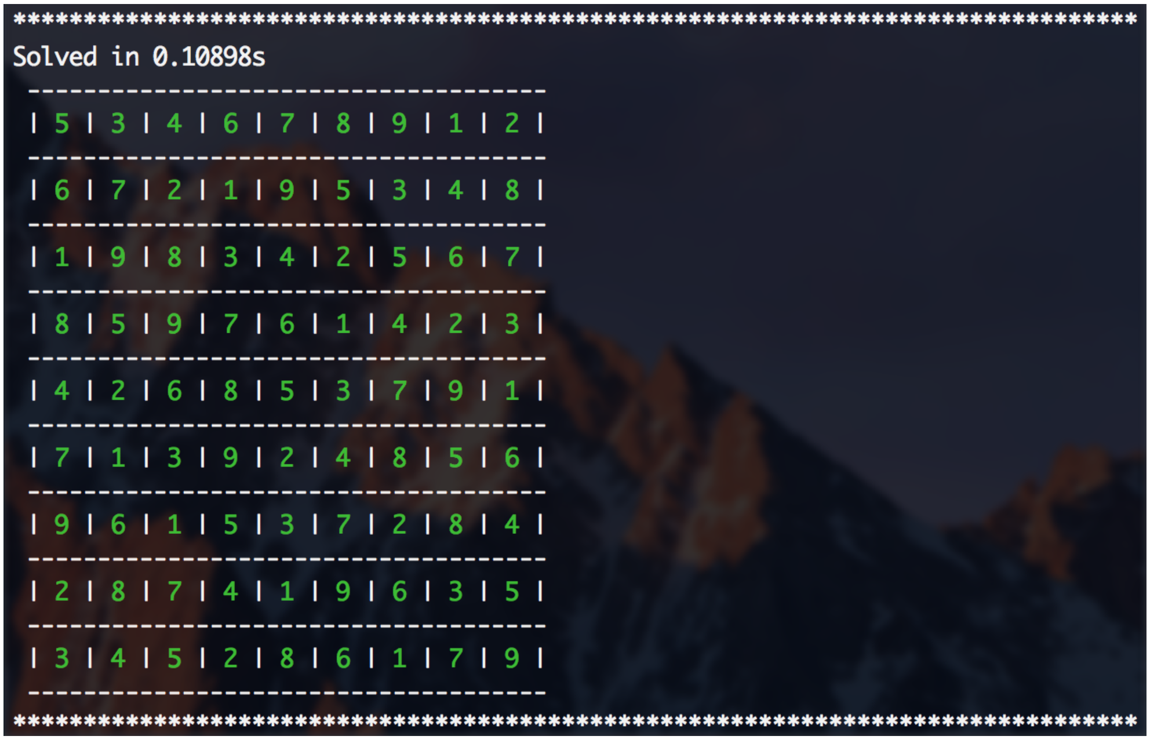
1.1 The Problem

Sudoku is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 9x9 grid with digits so that each column, each row, and each of the nine 3x3 sub-grids that compose the grid (also called boxes, blocks, regions, or sub-squares) contains all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a unique solution. (Wikipedia). The problem of solving a large sudoku board is NP-Complete and therefore would not be possible to brute force a solution. Since each column, row and sub-gird must contain all numbers 1 through 9 with no duplicates the space can be significantly reduced by thinking of the problem as a constraint satisfaction problem. While this still does not solve the problem of sudoku it does greatly reduce the time it takes to find a solution for smaller boards.



1.2 Constraint propagation

In order to find a solution to all sudoku boards I implemented 2 types of constraint propagation. Initially forward checking is used to eliminate variables from the domains of other variables in shared regions of the board (columns, rows, boxes). After forward checking is complete the AC3 arc-consistancy algorithm is used where each variable is connected to all other variables with shared domains through arcs. Each of these arcs are placed on a queue and checked one by one revising the domains of the other connected variables. When an arc is detected that has not been followed it is placed on the queue eventually when all arcs are followed the queue is empty and a solution may be found.



```
********************************************************************************
Solved in 0.10898s
 -------------------------------------
 | 5 | 3 | 4 | 6 | 7 | 8 | 9 | 1 | 2 |
 -------------------------------------
 | 6 | 7 | 2 | 1 | 9 | 5 | 3 | 4 | 8 |
 -------------------------------------
 | 1 | 9 | 8 | 3 | 4 | 2 | 5 | 6 | 7 |
 -------------------------------------
 | 8 | 5 | 9 | 7 | 6 | 1 | 4 | 2 | 3 |
 -------------------------------------
 | 4 | 2 | 6 | 8 | 5 | 3 | 7 | 9 | 1 |
 -------------------------------------
 | 7 | 1 | 3 | 9 | 2 | 4 | 8 | 5 | 6 |
 -------------------------------------
 | 9 | 6 | 1 | 5 | 3 | 7 | 2 | 8 | 4 |
 -------------------------------------
 | 2 | 8 | 7 | 4 | 1 | 9 | 6 | 3 | 5 |
 -------------------------------------
 | 3 | 4 | 5 | 2 | 8 | 6 | 1 | 7 | 9 |
 -------------------------------------
********************************************************************************
```

The possible solution is forward checked and if it is valid the solution is returned. If the solution is invalid then the entire board is copied into a stack with a single variable changed to a random guess. The process is then restarted and continues until a solution is found. The entire board is saved on the stack at each iteration to allow for back propagation should a guess cause a worse outcome.

2 Code

```python
#!/usr/bin/python3
#Sudoku Solver -- Robert Morouney Nov 09 2016 -- 069001422
#Usage: ./sudoku.py '/path/to/file' || cat /path/to/file | ./sudoku.py
import math
BOARD_SIZE = 9
BOX_SIZE = int(math.sqrt(BOARD_SIZE))
LINE_SIZE = 80


class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y


def all_points(size):
    return [Point(i, j) for i in range(0, size) for j in range(0, size)]


def compare(p1, p2):
    return (p1.x == p2.x) and (p1.y == p2.y)


class Var:
    def __init__(self, x, y, value=None):
        self.domain = list(range(1, BOARD_SIZE +
                                 1)) if value == None else [value]
        self.arcs = self._generate_arcs(x, y)

    def _generate_arcs(self, x, y):
        box = lambda x: range((x // BOX_SIZE) * BOX_SIZE, \
        (x // BOX_SIZE) * BOX_SIZE + BOX_SIZE)
        return [
            Point(i, j) \
            for i in range(0, BOARD_SIZE) for j in range(0, BOARD_SIZE) \
            if (i != x or y != j \
                ) and (i == x or j == y or (i in box(x) and j in box(y))) \
        ]


import fileinput


```

```
class Sudoku:
    def __init__(self):
        self.v = [[None for i in range(0, BOARD_SIZE)]
                   for j in range(0, BOARD_SIZE)]
        for i, line in enumerate(fileinput.input()):
            for j, c in enumerate(line[:].strip()):
                self.v[i][j] = Var(i, j, None if c == '_' else int(c))
        print('*' * LINE_SIZE + '\nSUDOKU SOLVER')
        self.print_board()
        print('*' * LINE_SIZE)

    def dom(self, p):
        return self.v[p.x][p.y].domain

    def set_dom(self, p, val):
        self.v[p.x][p.y].domain = [val]

    def arc(self, p):
        return self.v[p.x][p.y].arcs

    def get_arcs(self):
        arcs = []
        for p in all_points(BOARD_SIZE):
            for arc in self.arc(p):
                arcs.append((p, arc))
        return arcs

    def revise(self, v1, v2):
        removed = False
        d1 = self.dom(v1)
        d2 = self.dom(v2)
        for x in d1[:]:
            if not any([(x != y) for y in d2]):
                d1.remove(x)
                removed = True
        return removed

    def print_board(self):
        print("", "-" * 4 * BOARD_SIZE + "-")
        for i in range(0, BOARD_SIZE):
            for j in range(0, BOARD_SIZE):
                print(
                    " |",
                    color('_',31) if len(self.dom(Point(i, j))) > 1 else
                    color(str(self.dom(Point(i, j))[0]), 32),
                    end='')
            print(" |")
            print("", "-" * 4 * BOARD_SIZE + "-")

def color( c , code ):
    if os.name == 'nt':
        return c
    else:
        return "\033[0;"+str(code)+"m" + c + "\033[0m"

def ac_3(board):
    queue = board.get_arcs()
    with open('queue.out', 'w') as fout:
        while queue:
            fout.write("{}\n".format(len(queue)))
            i, j = queue.pop()
            if board.revise(i, j):
                for a in board.arc(i):
                    if not compare(i, a):
                        queue.append((a, i))


def cols():
    c = []
    for i in range(0, BOARD_SIZE):
        for j in range(0, BOARD_SIZE):
            c.append(Point(i, j))
    return c


def rows():
    r = []
    for i in range(0, BOARD_SIZE):
        for j in range(0, BOARD_SIZE):
            r.append(Point(j, i))
    return r


def boxes():
    b = []
    for x in range(0, BOX_SIZE):
        for y in range(0, BOX_SIZE):
            for i in range(BOX_SIZE * y, BOX_SIZE * y + BOX_SIZE):
                for j in range(BOX_SIZE * x, BOX_SIZE * x + BOX_SIZE):
                    b.append(Point(i, j))
    return b


def solve(board):
    changed = True
    while changed:
        ac_3(board)
        changed = False
        for region in [cols(), rows(), boxes()]:
            domain = list(range(1, BOARD_SIZE + 1))
            for p in region:
                d1 = board.dom(p)
                if len(d1) == 1 and d1[0] in domain: domain.remove(d1[0])
            for d in domain:
                if sum(board.dom(p).count(d) for p in region) == 1:
                    for p in region:
                        dt = board.dom(k)
                        if dt.count(d) > 0:
                            dt = [d]
                    changed = True


import copy
import queue
```

```
def search(board):
    q = queue.LifoQueue()
    q.put(copy.deepcopy(board))
    while q:
        current = q.get()
        solve(current)
        if all([len(current.dom(p)) == 1 for p in all_points(BOARD_SIZE)]):
            return current
        if not any([len(current.dom(p)) == 0 for p in all_points(BOARD_SIZE)]):
            p = [
                lp for lp in all_points(BOARD_SIZE) if len(current.dom(lp)) > 1
            ]
            for d in current.dom(p[0]):
                nexxt = copy.deepcopy(current)
                nexxt.set_dom(p[0], d)
                q.put(nexxt)


import os


def clear():
    if os.name != 'posix':
        os.system('cls')
    else:
        os.system('clear')


import time
if (__name__ == "__main__"):
    clear()
    board = Sudoku()
    _ = input("Press <Enter> to solve the puzzel...")
    clear()
    print('*' * LINE_SIZE)
    board.print_board()
    print('*' * LINE_SIZE)
    t0 = time.time()
    answer = search(board)
    print("Solved in {0:.5}s".format((time.time() - t0)))
    answer.print_board()
    print('*' * LINE_SIZE)
```

Listing 2.1: Sudoku as a CSP