HOW TO RUN:

1.)

(a) Briefly describe a CSP where the variables have a finite domain. You may not use map Coloring.

Tic-tac-toe:
Variables: 9 squares.
Domain: Each square can contain either an X or an O
Constraint: In order to meet the constraints, you need three consecutive X's or three consecutive O's on the board (horizontally, vertically, or diagonally).

(b) Briefly describe a CSP where the variables have an infinite, but discrete, domain.

Three people kick a soccer ball until they hit a goal. The variables are player1, player2, player3. Each variable represents the number of kicks the player did until they hit a goal. The domain for each player is an infinite sequence like {1,2,3,4…}. The constraints are that the Player1 != Player2. Player2 has to be less than Player3.

(c) Briefly describe a CSP where the variables have a continuous domain.

Measure the length of 3 crocodiles in feet. Variables are the lengths of each crocodile in feet. The variables are Croc1, Croc2, Croc3. Continuous Domain is (0,23 ft]. Croc2 cannot be longer than Croc3. Constraints are Snake1 must be longer than Snake4.

2.

(a) Suppose you were to eliminate variable E. Which constraints are removed? New constraints are created on which variables?

Removed: r6(C, E), r7(E, F), r4(B,E), and r9(E, G) are removed
New Constraints: With variables B, C,F,G. r(B, G), r(C, F), r(C, B), r(C, G), r(G, F), r(B, F)

i. A CSP has a variable and each variable has a domain. For the Sudoku puzzle, what are the variable and what are their domains?

The variables: The 81 squares in a 9x9 grid.
The domain: any number from 1-9

ii. What data structure did you use to store your Sudoku Puzzle?

I used a couple of data structures. I used a list to store the coordinates of each box. I also used a dictionary to store the domains of each coordinate. I used an object to store every aspect/behavior of my representation of the sudoku puzzle.

iii. What constraints did you add for the puzzle? Provide snippet of your code where you define the constraints.

Given a cell, that cell cannot have a value that exists within its respectful row, columns, or 3x3 block of squares.

```
def getBoardConstraints(self):
    c1 = self.getColumnConstraints()#A1,B1,C1...
    c2 = self.getRowConstraints()#A1,A2,A3,A4...
    c3 = self.getBlockConstraints()#A1,A2,A3,B1,B2,B3,C1,C2,C3... (The 3x3 blocks)
    all_constraints = (c1+c2+c3)
    all_binary_constraints = []
    for constraint in all_constraints:
        binaries = []
        for binary in itertools.permutations(constraint,2):
            binaries.append(binary)
        for binary in binaries:
            arr = list(binary)
            if arr not in all_binary_constraints:
                all_binary_constraints.append([arr[0],arr[1]])
    return all_binary_constraints
```

iv. To implement AC3 you need to:
A. Check for Arc Consistency
B. Implement Arc Consistency
Describe how you implemented this and provide snippets of your code to support your answer.

Pop each constraint out of the queue. I check if the popped constraint has a conflict with the domains. The revise function checks to see a value has to removed from the domain of that square. It checks to see if the two values are equal. If they are equal, that value is removed from domain. Then the revise function returns true if value is removed from domain. If the revise function returns true, then it checks the length of domain. If the length of domain is 0 then the ac3 function returns false. If a value from the domain is removed from either squares, then I append the second squares coordinates into the queue. After checking all the neighbors, you go the next tuple by popping again from queue.

```python
def revise(sudoku, a, b):
    revised = False
    for x in sudoku.domains[a]:
        remove_element = True
        for y in sudoku.domains[b]:
            if x != y:
                remove_element = False
        if remove_element:
            revised = True
            sudoku.domains[a].remove(x)
    return revised
def ac3(sudoku_board):
    queue = list(sudoku_board.constraints)
    while queue:
        i,j = queue.pop(0)
        if revise(sudoku_board,i,j):
            if len(sudoku_board.domains[i]) == 0:
                return False
            for k in sudoku_board.neighbors[i]:
                if k != i:
                    queue.append([k,i])
    return True
```

MCV and LRV:

```python
# Most Constrained Variable
def mcv(assignment, sudoku):
    unassigned = []
    for v in sudoku.variables:
        if v not in assignment:
            unassigned.append(v)
    return min(unassigned, key=lambda var: len(sudoku.domains[var]))
# Least Constrained Value
def lcv(sudoku, var):
    if len(sudoku.domains[var]) == 1:
        return sudoku.domains[var]
    return sorted(sudoku.domains[var], key=lambda val: sudoku.getNumOfConflicts(sudoku, var, val))
```

For mcv you go through all the variables and see if it exists within a dictionary. It does not, then you append the variable to the list. Then I used a lambda function that return the variable with the minimum domain length.

For lcv I check to see if the length of the domain of the variable is 1. If it is 1 then I just return the domains of that variable. However if it is not one, then I return the sorted list of the domains of the puzzle by getting the number of conflicts of that variable.