Homework 2 – AI

Leonard Christopher Limanjaya – 20225087

1. Implement Sudoku solver with CSP and any suitable search methods.
   Variables (X): each empty cell on the sudoku board
   Domains (D): A set number (1-9) which already used in a row, column, or 3x3 squares
   Constraints (C): No repeating number in rows, columns, and the 3x3 squares

   I watch this video explanation for the reference of making this code and making my own approach:
   https://towardsdatascience.com/solving-nonograms-with-120-lines-of-code-a7c6e0f627e4

   How the code works:

a. Get input from the terminal with the following format:

```
$ sudoku  <- execute the program
0 4 0 0 0 0 0 0 0 <- user's input from keyboard
0 0 1 0 3 4 6 2 0
6 0 3 0 0 0 0 7 0
0 0 0 4 8 3 5 0 7
0 0 0 0 5 0 0 6 0
0 0 0 0 0 9 0 4 0
0 0 5 0 0 0 0 0 1
8 0 0 5 4 7 3 9 6
0 0 0 0 2 1 0 0 0
2 4 8 6 7 5 1 3 9 <- output from the program
7 5 1 9 3 4 6 2 8
6 9 3 2 1 8 4 7 5
9 2 6 4 8 3 5 1 7
1 8 4 7 5 2 9 6 3
5 3 7 1 6 9 8 4 2
4 7 5 3 9 6 2 8 1
8 1 2 5 4 7 3 9 6
3 6 9 8 2 1 7 5 4
```

The following function will get input from the terminal and store it to the list

```python

def get_input(self):
    """Getting Input from terminal
    Convert into list
    returns list of input
    """
    self.matrix = []
    for _ in range(9):
        self.matrix.append([int(num) for num in input().split(" ")])
```

b. Find any zero value on the sudoku board. The code will run solve() function to solve the sudoku problem.

```python
def solver(self):
    x, y = self.find_zero()
    if x == -1 and y == -1:
        return True
    for n in range(1, 10):
        if self.check_value(y, x, n):
            print("change")
            self.matrix[x][y] = n
            if self.solver():
                return True
            # self.solver()
            # print("nul")
            self.matrix[x][y] = 0
```

At first, this function will run find_zero() function to get the position of 0 value as the return value. This function will go through the entire list and search where is the 0. If there is no 0 value in the list, then the function will return -1, -1

```python
def find_zero(self):
    for x in range(9):
        for y in range(9):
            if self.matrix[x][y] == 0:
                return x, y
    return -1, -1
```

c. Try to choose the number that available on that point

```python
def solver(self):
    x, y = self.find_zero()
    if x == -1 and y == -1:
        return True
    for n in range(1, 10):
        if self.check_value(y, x, n):
            print("change")
            self.matrix[x][y] = n
            if self.solver():
                return True
            # self.solver()
            # print("nul")
            self.matrix[x][y] = 0
```

In this step the code will try to fill the empty one with 1-10, and the program will check if the value is available or not from the check_value() function. If it is available, the code will assign that number to the sudoku matrix. And the program will continue to run solver() function recursively. The following picture will show the code to check_value

```python
def check_columns(self, col, num):
    for row in range(9):
        if self.matrix[row][col] == num:
            return False
    return True

def check_rows(self, row, num):
    for col in range(9):
        if self.matrix[row][col] == num:
            return False
    return True

def check_box(self, col, row, num):
    box_col = col - col%3
    box_row = row - row%3
    for c in range(box_col, box_col+3):
        for r in range(box_row, box_row+3):
            if self.matrix[r][c] == num:
                return False
    return True

def check_value(self, col , row, num):
    if self.check_box(col, row, num) and self.check_columns(col, num) \
        and self.check_rows(row, num):
        return True
    return False

def find_zero(self):
    for x in range(9):
        for y in range(9):
            if self.matrix[x][y] == 0:
                return x, y
    return -1, -1

```

d. If at some point it can not continue to fill until no zero left, the code will go back to the previous step/tiles

e. Repeat steps b – d until no remaining zero left. This happened when the find_zero() function return -1,1 and the solve() function will return False.

f. Print the solved sudoku

2. Implement a Nonogram solver with CSP and any suitable search methods.
Variables (X): start cells of row or columns segments
Domains (D): -1, 0, and 1. -1 denoted as white, 1 as black, and 0 as unknown
Constraints (C): Relationship between adjacent segments, each segment will have black space(s) between them if the segments is more than 2

I read this article to understand the algorithm how to solve this problem, and I make my own approach:
https://towardsdatascience.com/solving-nonograms-with-120-lines-of-code-a7c6e0f627e4

How the code works

a. Get input from the terminal with the following format:

```
$ nonogram  <- execute the program
10 10 <- height and width
1 1 <- from row 1
2 2
5
1 1 1
7
5 2
3 1
3 1
4 2
7
1 <- column 1
6
2 6
8
2 6
6 2
1 1
1
1 2
4
 *   *     <- output from the program
 ** **
 *****
 * * *
*******
 *****  **
  ***    *
  ***    *
  ****  **
  *******
```

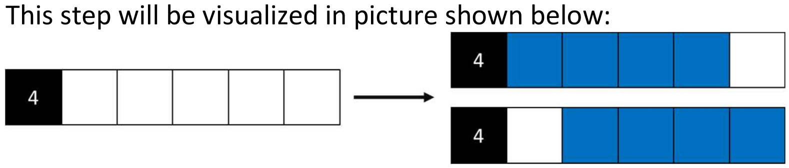
The following picture is the code to get the input from terminal

```python

def get_input(self):
    size = input("Get").split(" ")
    self.row_size = int(size[0])
    self.col_size = int(size[1])
    for i in range(self.row_size):
        self.row[i]=[int(num) for num in input().split(" ")]

    for i in range(self.col_size):
        self.col[i]=[int(num) for num in input().split(" ")]

    for _ in range(self.col_size):
        self.nonogram.append(["0"] * self.col_size)
        self.union.append([0] * self.col_size)

```
From the terminal input, the code will be store on the variable with list format

b. First, after the program gets the input, the program will generate the possible combination of each row and column. To make this, this program will utilize itertools library to make the combination. These three functions will help to make the combination.

```python

def get_combinations(self, input_l, max_length):
    empty = max_length - (sum(input_l) + len(input_l) - 1)
    opts = combinations(range(len(input_l) + empty), len(input_l))
    opt_lists = []
    for opt in opts:
        opt_list = []
        for p in opt:
            opt_list.append(p)
        opt_lists.append(opt_list)
    return opt_lists

def generate_actual_combinations(self, combs, lists, size):
    all_of_it = []
    # print(combs)
    for c in combs:
        actual = [0] * size
        start = 0
        for n in range(len(c)):
            for k in range(lists[n]):
                actual[start+c[n]+k] = 1
            start = start + lists[n]
        all_of_it.append(actual)
    return all_of_it

def generate_combinations(self):
    for row in self.row:
        self.opts_row[row]=self.generate_actual_combinations(
            self.get_combinations(self.row[row], self.row_size),
            self.row[row], self.row_size)

    for col in self.col:
        self.opts_col[col]=self.generate_actual_combinations(
            self.get_combinations(self.col[col], self.col_size),
            self.col[col], self.col_size)

```

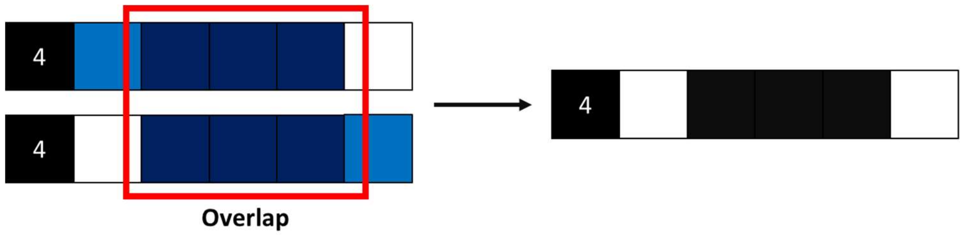
This step will be visualized in picture shown below:



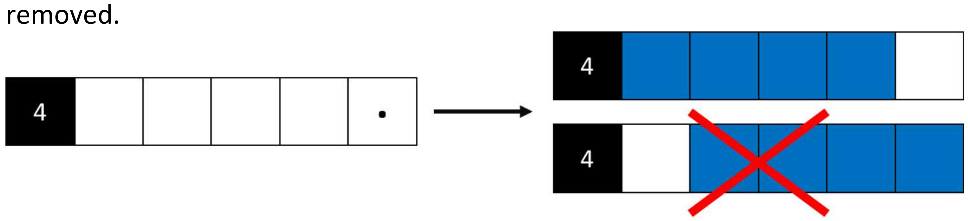
c. After that, the program gets the overlap tiles from each row and column. This process included the black tiles and white tiles.

```
def cross_validate(self):
    for row in range(self.row_size):
        self.union_row[row]=self.check_overlap(self.opts_row[row],
                                               [sum(i) for i in \
                                                zip(*self.opts_row[row])])
        for opts in self.opts_row[row]:
            for col in range(self.col_size):

                if self.union[row][col] == 1 and opts[col] != 1:
                    self.opts_row[row].remove(opts)
                    break

                elif self.union[row][col] == -1 and opts[col] != 0:
                    self.opts_row[row].remove(opts)
                    break

        for col in range(len(self.union_row[row])):
            if self.union_row[row][col] == 1:
                self.nonogram[row][col] = "*"
                self.union[row][col] = 1
            if self.union_row[row][col] == -1:
                self.nonogram[row][col] = " "
                self.union[row][col] = -1

    for col in range(self.col_size):
        self.union_col[col]=self.check_overlap(self.opts_col[col],
                                               [sum(i) for i in \
                                                zip(*self.opts_col[col])])
        for opts in self.opts_col[col]:
            for row in range(self.row_size):

                if self.union[row][col] == 1 and opts[row] != 1:
                    self.opts_col[col].remove(opts)
                    break
                elif self.union[row][col] == -1 and opts[row] != 0:
                    self.opts_col[col].remove(opts)
                    break

        for row in range(len(self.union_col[col])):
            if self.union_col[col][row] == 1:
                self.nonogram[row][col] = "*"
                self.union[row][col] = 1
            if self.union_col[col][row] == -1:
                self.nonogram[row][col] = " "
                self.union[row][col] = -1
```

This picture below will visualized this step:



d. The program will check if the possible tiles in each row violate known tiles (black and white). If the possible rows or columns violate the known tiles, the possible rows or columns will be removed.



The below possible tiles are violating the known tiles, so this possible row will be removed.

e. The program will start again from step 2 until the remaining possible rows or columns in each row and column is 1.
f. In the end, the code will print the result, 1 will represent “*”, and -1 will be represented by “ “.