EAST WEST UNIVERSITY

Student's Name	Student ID
Md. Ashiqur Rahman	2020-1-60-173
Md Tahsin	2020-2-60-112
Navid Zaman	2020-2-60-044
Adnan Mahmud	2018-3-60-023

Department of Computer Science and Engineering
Course Title: Artificial Intelligence
Course Code: CSE366
Section: 03

Submitted To

Dr Md Rifat Ahmmad Rashid

Assistant Professor

Submission Date

January 01, 2023

❖ Problem Statement

Develop a game of Sudoku solver. Following these three approaches -

- ○ By using Genetic algorithm
- ○ Use alpha beta pruning to solve the puzzle.
- ○ Solve as a constraint satisfaction problem.

❖ Content

Introduction: In classic Sudoku game, the objective is to fill a 9 × 9 grid with numbers that compose each column, each row, and each grid of nine 3 × 3 sub grids called "blocks", One contains all the numbers from 1 to 9. The puzzle setter provides a partially complete grid.

Algorithm details: here we used two approaches to solve this sudoku game.

1.Genetic Algorithm

2. Constraint satisfaction problem

Genetic Algorithm : The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. Genetic Algorithms are inspired by the Natural selection, Darwin's theory of evolution, according to which the natural elimination of the least able individuals in the "struggle for life" allows the species to improve from generation to generation.

Here are the steps to follow:

1.Build a population of potential solutions of the problem and evaluate them according to a scoring method.

2.Select best elements to become the "parents" of new solutions "children" who will have elements of the solution "father" and the solution "mother".

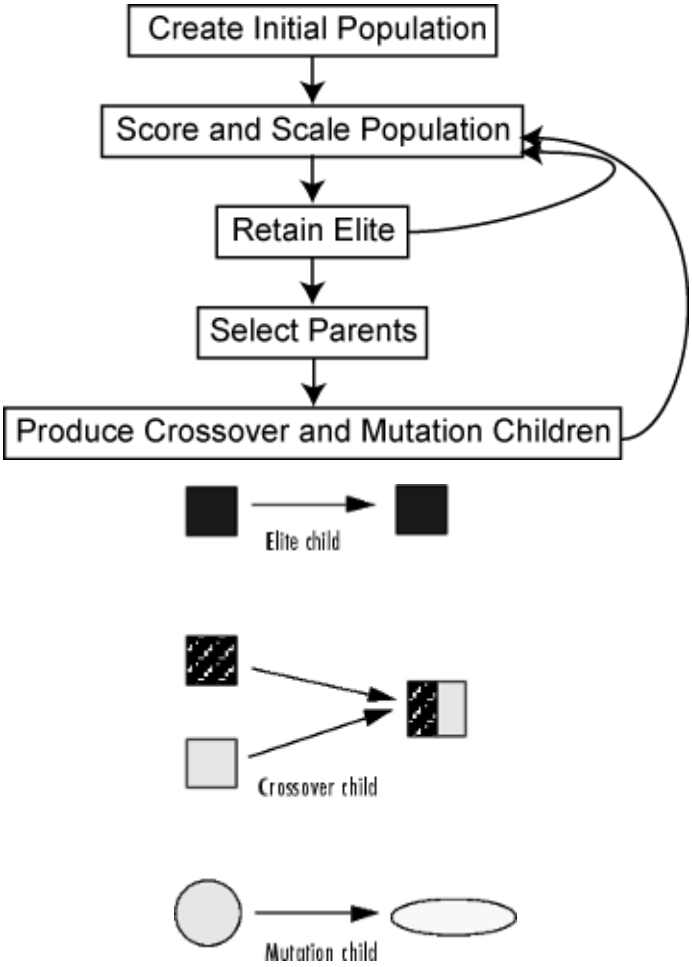
3.To avoid eugenics, a certain part of the population (including the bad solutions) is even selected to be parents.

3.Finally, a dose of randomness is integrated by mutating a certain part of the new population generated.

Then the cycle begins again with this new population which is then evaluated to serve as parents to the next.

4.And so on until the problem is solved.

These steps can be represented as flowchart simply as following –



Approach to solve Sudoku problem using Genetic algorithm:

In terms of solving 9X9 SUDOKU board solving, we have generated 10000 population at first according to the selected board and calculated their fitness score. Here, the fitness score is the number of duplicate values in rows and columns. After generating the fitness score, we have ranked them according to their fitness score. Among the overall population 2 population is chosen randomly as parents and child will be generated from them. Then we applied mutation process where 2 random cells is swapped within a population, this overall process continues until a population is found to have fitness score of 0 or it won't be able to solve the sudoku within expected generation.

- Create the first population: Here we just randomly generate some 'solutions' to evaluate. If we want our algorithm to converge, we have to help it a little bit so the creation of one individual is not totally random but of course, there can be duplicates in rows and columns in the generated populations.
- Ranking: This is the most important thing to think about when dealing with GA because this will determine how we will evaluate our solutions, so which ones will be selected. In our case the fitness function counts duplicates in the potential solution.So our objective is to minimize this score: the best individuals have a low fitness value .
- Create a child: Two potential solutions are chosen randomly and generate n children (that n is a parameter to set which denotes the number of child will be generated from the parents).What we have done is we took some elements from the father and others from the mother.
- Mutation: A certain percentage of the population will mutate .Applied to the sudoku solving problem, we chose to randomly pick one grid and, within this grid, randomly swap 2 values. The swap is made in a way that it is not allowed to swap a 'fixed value', i.e a value that is known and provided).With such a mutation approach we can guarantee that grid remains coherent and without duplicates but fitness evaluation of a 'mutated' child is different than the one from the same element before its mutation. It can have improved it or not.

Constraint satisfaction problem: Constraint Satisfaction Problem (CSP) is a problem in which we need to find a solution that satisfies a set of constraints. A CSP consists of a set of variables, each with a domain of possible values, and a set of constraints that specify which combinations of values are allowed. The goal is to find an assignment of values to the variables such that all the constraints are satisfied. A constraint satisfaction problem (CSP) is a problem that requires its solution within some limitations or conditions also known as constraints. It consists of the following:

A finite set of variables which stores the solution (V = {V1, V2, V3,....., Vn})

A set of discrete values known as domain from which the solution is picked (D = {D1, D2, D3,.....,Dn})

A finite set of constraints (C = {C1, C2, C3,....., Cn})

The elements in the domain can be both continuous and discrete but in AI, we generally only deal with discrete values.

Approach to solve Sudoku problem using CSP algorithm:

Each empty cell is considered as variables and the domains can be numbers from (1 to 9) according to the constraints ,which basically numbers that can fill in the cell considering the rows, columns and 3x3 cell.It is a intelligent backtracking algorithm which checks all the cells with possible value and finds solution.

A CSP is specified by the following inputs:

variables	A list of variables; each is atomic (e.g. int or string).
domains	A dictionary of {var:[possible_value, ...]} entries.
neighbors	A dictionary of {var:[var,...]} that for each variable lists the other variables that participate in constraints.
constraints	A function f(A, a, B, b) that returns true if neighbors A, B satisfy the constraint when they have values A=a, B=b

Pseudo Code:

Pseudo code for genetic algorithm:

```
generate initial population
repeat
    rank the solutions, and retain only the percentage specified by selection rate
    repeat
        randomly select two solutions from the population
        randomly choose a crossover point
        recombine the solutions to produce n new solutions
        apply the mutation operator to the solutions
    until a new population has been produced
until a solution is found or the maximum number of generations is reached
```

Pseudo code for CSP algorithm:

```
function csp_sudoku_solver(puzzle):
    variables = initialize_variables(puzzle)
    domains = initialize_domains(puzzle)

    constraints = initialize_constraints(variables)
    solution = backtrack(variables, domains, constraints)
    return solution


function backtrack(variables, domains, constraints):
    if all_variables_assigned(variables):
        if constraints_satisfied(variables, constraints):
            return variables
        else:
            return failure

    variable = select_unassigned_variable_mrv(variables, domains)

    for value in domain[variable]:
        variables_copy = deep_copy(variables)
        domains_copy = deep_
```

Python Implementation:

Class: main.py

```
from tkinter import *
from CSP_classes.gui import SudokuUI
root = Tk()

SudokuUI(root)

root.title("Sudoku Solver")
root.mainloop()
```

Class: gui.py

```
MARGIN = 20  # Pixels around the board
SIDE = 50  # Width of every board cell
WIDTH_B = HEIGHT_B = MARGIN * 2 + SIDE * 9  # Width and height of the whole board
WIDTH = WIDTH_B + 180  # Width of board and buttons solve and reset


class SudokuUI(Frame):

    def __init__(self, parent):
        self.parent = parent
        # we start with a blank board
        self.original_board = [[0 for j in range(9)] for i in range(9)]
        # ofc we should have another board in which we will show solution
        self.current_board = copy.deepcopy(self.original_board)
        Frame.__init__(self, parent)
        self.row, self.col = 0, 0
        self.__initUI()

    def __initUI(self):
        # we will initializate stuff that will be shown in the gui
        self.pack(fill=BOTH, expand=1)
        self.canvas = Canvas(self, width=WIDTH_B, height=HEIGHT_B)
        self.canvas.pack(fill=BOTH, side=TOP)
        self.canvas.grid(row=0, column=0, rowspan=30, columnspan=60)

        # level will be used to select the lvl of the board, 1 means easy and 2 hard
        self.level = IntVar(value=1)
        # which will be used to select which board at which lvl, there are 3 for each level
        self.which = 0

        # we will need a StringVar so that the client can see the time used by an algorithm
        self.time = StringVar()
        self.time.set("Time:                    ")
        # same for number of backtracks
        self.n_bt = StringVar()
        self.n_bt.set("N. BT:   ")

        self.make_menu()

        # the default will be the board of lvl 1 and which 1
        self.__change_level()
        Label(self, text="Instructions").grid(row=1, column=61)
        Label(self, text="1.Select Difficulty Level").grid(row=2, column=61)
        Label(self, text="2.Select Level").grid(row=3, column=61)
        Label(self, text="3.Select Algorithm & done!").grid(row=4, column=61)
        self.clear_button = Button(self, text="Reset", command=self.__clear_board, width=15,
height=2)
        self.clear_button.grid(row=10, column=61, padx=20, columnspan=3)
        self.solve_button = Button(self, text="Solve with CSP", command=self.solve_clicked,
```

```
width=15, height=2)
        self.solve_button.grid(row=13, column=61, padx=20, columnspan=3)
        self.solve_button = Button(self, text="Solve with G.A", command=self.solve_clicked_GA,
width=15, height=2)
        self.solve_button.grid(row=14, column=61, padx=20, columnspan=3)

        lbltime = Label(self, textvariable=self.time)
        lblBT = Label(self, textvariable=self.n_bt)

        #Label(self, text="Inference:              ").grid(row=14, column=61)
        lbltime.grid(row=30, column=0)

        lblBT.grid(row=32, column=0)
        self.inference = StringVar()
        self.radio = []
        self.radio.append(Radiobutton(self, text="", variable=self.inference,
value="NO_INFERENCE"))
        self.radio[0].grid(row=15, column=62, padx=2)

        self.inference.set("NO_INFERENCE")

        Label(self, text="Select Difficulty :").grid(row=18, column=61)
        lbltime.grid(row=30, column=0)
        lblBT.grid(row=32, column=0)
       # self.which = IntVar()
        self.radio2 = []
        self.radio2.append(Radiobutton(self, text="Level 1", variable=self.which, value=0))
        self.radio2[0].grid(row=18, column=62, padx=2)

        self.radio2.append(Radiobutton(self, text="Level 2", variable=self.which, value=1))
        self.radio2[1].grid(row=19, column=62, padx=2)

        self.radio2.append(Radiobutton(self, text="Level 3", variable=self.which, value=2))
        self.radio2[2].grid(row=20, column=62, padx=2
        self.var_to_choose = "MRV"

        self.__draw_grid()
        self.__draw_puzzle()

    def solve_clicked(self):

        # we are searching for a solution so it is good to disable buttons
        for rb in self.radio:
            rb.config(state=DISABLED)
        self.clear_button.config(state=DISABLED)
        self.solve_button.config(state=DISABLED)
        self.menu_bar.entryconfig("Level", state="disabled")
        p = threading.Thread(target=self.solve_sudoku)
        p.start()
        messagebox.showinfo("Working", "We are looking for a solution, please wait some seconds
...")

    def solve_clicked_GA(self):
       p = threading.Thread(target=self.GANA)
       p.start()
       messagebox.showinfo("Working", "We are looking for a solution, please wait some seconds
...")

    def GANA(self):
        population_size = 10000
        selection_rate = 0.2
        random_selection_rate = 0.2
        nb_children = 5
        mutation_rate = 0.3
        max_nb_generations = 500
        presolving = False
        model_to_solve ="3x3-easy-02"
        restart_after_n_generations_without_improvement = 40

        if self.level == 1:
```

```
            if self.which == 0:
                model_to_solve="easy_0"
            elif self.which == 1:
                model_to_solve = "easy_1"
            elif self.which == 2:
                model_to_solve = "easy_2"
        elif self.level == 2:
            if self.which == 0:
                model_to_solve = "hard_0"
            elif self.which == 1:
                model_to_solve = "hard_1"
            elif self.which == 2:
                model_to_solve = "hard_2"

        sga = SudokuGA(population_size, selection_rate, random_selection_rate, nb_children,
max_nb_generations,
                       mutation_rate, model_to_solve, presolving,
restart_after_n_generations_without_improvement)

        sga.run()
        for i in range(9):
            for j in range(9):
                self.current_board[i][j] = classes.sudoku.inner[i][j]
        self.__draw_puzzle()

    def solve_sudoku(self):

        s = SudokuCSP(self.current_board)
        inf, dv, suv = None, None, None

        if self.inference.get() == "NO_INFERENCE":
            inf = no_inference
        elif self.inference.get() == "FC":
            inf = forward_checking
        elif self.inference.get() == "MAC":
            inf = mac

        if self.var_to_choose == "MRV":
            suv = mrv

        start = timer()
        a = backtracking_search(s, select_unassigned_variable=suv,
order_domain_values=unordered_domain_values,
                                inference=inf)
        end = timer()
        # if a isn't null we found a solution so we will show it in the current board
        # if a is null then we send a message to the user that the initial board
        # breaks some constraints
        if a:
            for i in range(9):
                for j in range(9):
                    index = i * 9 + j
                    self.current_board[i][j] = a.get("CELL" + str(index))
        else:
            messagebox.showerror("Error", "Invalid sudoku puzzle, please check the initial
state")

        # showing solution
        self.__draw_puzzle()
        self.time.set("Time: "+str(round(end-start, 5))+" seconds")
        self.n_bt.set("N. BR: "+str(s.n_bt))

        # re-enabling buttons for search a new solution
        for rb in self.radio:
            rb.config(state=NORMAL)
        self.clear_button.config(state=NORMAL)
        self.solve_button.config(state=NORMAL)
        self.menu_bar.entryconfig("Level", state="normal")

    def make_menu(self):
```

```
        # creating menu with level Easy and Hard
        self.menu_bar = Menu(self.parent)
        self.parent.configure(menu=self.menu_bar)
        level_menu = Menu(self.menu_bar, tearoff=False)
        self.menu_bar.add_cascade(label="Level", menu=level_menu)
        level_menu.add_radiobutton(label="Easy", variable=self.level, value=1,
command=self.__change_level)
        level_menu.add_radiobutton(label="Hard", variable=self.level, value=2,
command=self.__change_level)

    def __change_level(self):
        # to add a new board, you just have to change %3 to %4 and then add another
        # clause elif like "elif self.which == 3:"
        self.which = (self.which+1) % 3
        if self.level.get() == 1:
            if self.which == 0:
                self.original_board[0] = [0, 6, 0, 3, 0, 0, 8, 0, 4]
                self.original_board[1] = [5, 3, 7, 0, 9, 0, 0, 0, 0]
                self.original_board[2] = [0, 4, 0, 0, 0, 6, 0, 0, 7]
                self.original_board[3] = [0, 9, 0, 0, 5, 0, 0, 0, 0]
                self.original_board[4] = [0, 0, 0, 0, 0, 0, 0, 0, 0]
                self.original_board[5] = [7, 1, 3, 0, 2, 0, 0, 4, 0]
                self.original_board[6] = [3, 0, 6, 4, 0, 0, 0, 1, 0]
                self.original_board[7] = [0, 0, 0, 0, 6, 0, 5, 2, 3]
                self.original_board[8] = [1, 0, 2, 0, 0, 9, 0, 8, 0]
            elif self.which == 1:
                self.original_board[0] = [7, 9, 0, 4, 0, 2, 3, 8, 1]
                self.original_board[1] = [5, 0, 3, 0, 0, 0, 9, 0, 0]
                self.original_board[2] = [0, 0, 0, 0, 3, 0, 0, 7, 0]
                self.original_board[3] = [0, 0, 0, 0, 0, 5, 0, 0, 2]
                self.original_board[4] = [9, 2, 0, 8, 1, 0, 7, 0, 0]
                self.original_board[5] = [4, 6, 0, 0, 0, 0, 5, 1, 9]
                self.original_board[6] = [0, 1, 0, 0, 0, 0, 2, 3, 8]
                self.original_board[7] = [8, 0, 0, 0, 4, 1, 0, 0, 0]
                self.original_board[8] = [0, 0, 9, 0, 8, 0, 1, 0, 4]
            elif self.which == 2:
                self.original_board[0] = [0, 8, 0, 0, 0, 0, 0, 9, 0]
                self.original_board[1] = [0, 0, 7, 5, 0, 2, 8, 0, 0]
                self.original_board[2] = [6, 0, 0, 8, 0, 7, 0, 0, 5]
                self.original_board[3] = [3, 7, 0, 0, 8, 0, 0, 5, 1]
                self.original_board[4] = [2, 0, 0, 0, 0, 0, 0, 0, 8]
                self.original_board[5] = [9, 5, 0, 0, 4, 0, 0, 3, 2]
                self.original_board[6] = [8, 0, 0, 1, 0, 4, 0, 0, 9]
                self.original_board[7] = [0, 0, 1, 9, 0, 3, 6, 0, 0]
                self.original_board[8] = [0, 4, 0, 0, 0, 0, 0, 2, 0]

        elif self.level.get() == 2:
            if self.which == 0:
                self.original_board[0] = [0, 0, 0, 8, 0, 0, 9, 0, 0]
                self.original_board[1] = [0, 9, 0, 0, 7, 0, 0, 0, 4]
                self.original_board[2] = [0, 8, 4, 0, 0, 0, 0, 6, 0]
                self.original_board[3] = [0, 0, 0, 4, 1, 0, 2, 0, 0]
                self.original_board[4] = [0, 0, 3, 0, 0, 0, 5, 0, 0]
                self.original_board[5] = [0, 0, 1, 0, 6, 9, 0, 0, 0]
                self.original_board[6] = [0, 2, 0, 0, 0, 0, 7, 4, 0]
                self.original_board[7] = [9, 0, 0, 0, 2, 0, 0, 3, 0]
                self.original_board[8] = [0, 0, 7, 0, 0, 6, 0, 0, 0]
            elif self.which == 1:
                self.original_board[0] = [0, 0, 9, 7, 0, 0, 0, 0, 3]
                self.original_board[1] = [0, 0, 4, 5, 0, 0, 0, 9, 0]
                self.original_board[2] = [1, 5, 0, 0, 9, 0, 0, 0, 0]
                self.original_board[3] = [0, 0, 0, 0, 0, 1, 0, 0, 0]
                self.original_board[4] = [0, 0, 0, 0, 8, 0, 0, 6, 1]
                self.original_board[5] = [0, 0, 2, 0, 0, 0, 8, 0, 0]
                self.original_board[6] = [0, 6, 0, 2, 0, 4, 0, 0, 0]
                self.original_board[7] = [0, 0, 0, 0, 0, 0, 7, 0, 0]
                self.original_board[8] = [7, 9, 0, 0, 0, 0, 0, 5, 0]
            elif self.which == 2:
                self.original_board[0] = [0, 0, 9, 7, 0, 0, 0, 0, 3]
                self.original_board[1] = [0, 0, 4, 5, 0, 0, 0, 9, 0]
```

```
                self.original_board[2] = [1, 5, 0, 0, 9, 0, 0, 0, 0]
                self.original_board[3] = [0, 0, 0, 0, 0, 1, 0, 0, 0]
                self.original_board[4] = [0, 0, 0, 0, 8, 0, 0, 6, 1]
                self.original_board[5] = [0, 0, 2, 0, 0, 0, 8, 0, 0]
                self.original_board[6] = [0, 6, 0, 2, 0, 4, 0, 0, 0]
                self.original_board[7] = [0, 0, 0, 0, 0, 0, 7, 0, 0]
                self.original_board[8] = [7, 9, 0, 0, 0, 0, 0, 5, 0]

        self.current_board = copy.deepcopy(self.original_board)

        self.__draw_puzzle()

    def __draw_grid(self):

        for i in range(10):
            if i % 3 == 0:
                color = "black"
            else:
                color = "gray"
            x0 = MARGIN + i * SIDE
            y0 = MARGIN
            x1 = MARGIN + i * SIDE
            y1 = HEIGHT_B - MARGIN
            self.canvas.create_line(x0, y0, x1, y1, fill=color)
            x0 = MARGIN
            y0 = MARGIN + i * SIDE
            x1 = WIDTH_B - MARGIN
            y1 = MARGIN + i * SIDE
            self.canvas.create_line(x0, y0, x1, y1, fill=color)

    def __draw_puzzle(self):
        self.canvas.delete("numbers")
        self.time.set("Time:                  ")
        self.n_bt.set("N. BT:   ")
        for i in range(9):
            for j in range(9):
                cell = self.current_board[i][j]
                if cell != 0:
                    x = MARGIN + j * SIDE + SIDE / 2
                    y = MARGIN + i * SIDE + SIDE / 2
                    if str(cell) == str(self.original_board[i][j]):
                        self.canvas.create_text(x, y, text=cell, tags="numbers", fill="black")
                    else:
                        self.canvas.create_text(x, y, text=cell, tags="numbers", fill="red")

    def __clear_board(self):
        self.current_board = copy.deepcopy(self.original_board)
        self.__draw_puzzle()
```

Class: sudoku_genetics.py

```
class SudokuGA(object):
    _population_size = None
    _selection_rate = None
    _random_selection_rate = None
    _nb_children = None
    _max_nb_generations = None
    _mutation_rate = None
    _model_to_solve = None
    _presolving = None
    _restart_after_n_generations_without_improvement = None
    _start_time = None

    def __init__(self, population_size, selection_rate, random_selection_rate, nb_children,
max_nb_generations,
                 mutation_rate, model_to_solve, presolving,
restart_after_n_generations_without_improvement):
```

```
        self._population_size = population_size
        self._selection_rate = selection_rate
        self._random_selection_rate = random_selection_rate
        self._nb_children = nb_children
        self._max_nb_generations = max_nb_generations
        self._mutation_rate = mutation_rate
        self._model_to_solve = model_to_solve
        self._presolving = presolving
        self._restart_after_n_generations_without_improvement =
restart_after_n_generations_without_improvement

    def run(self):
        values_to_set = self._load().get_initial_values() #text reading

        best_data = []
        worst_data = []
        found = False
        overall_nb_generations_done = 0
        restart_counter = 0

        while overall_nb_generations_done < self._max_nb_generations and not found:
            new_population = ga_utils.create_generation(self._population_size, values_to_set)

            nb_generations_done = 0
            remember_the_best = 0
            nb_generations_without_improvement = 0

            # Loop until max allowed generations is reached or a solution is found
            while nb_generations_done < self._max_nb_generations and not found:
                # Rank the solutions
                ranked_population = ga_utils.rank_population(new_population)
                best_solution = ranked_population[0]
                best_score = best_solution.fitness()
                worst_score = ranked_population[-1].fitness()
                best_data.append(best_score)
                worst_data.append(worst_score)

                # Manage best value and improvements among new generations over time
                if remember_the_best == best_score:
                    nb_generations_without_improvement += 1
                else:
                    remember_the_best = best_score
                if 0 < self._restart_after_n_generations_without_improvement <
nb_generations_without_improvement:
                    print("No improvement since {} generations, restarting the program".
                          format(self._restart_after_n_generations_without_improvement))
                    restart_counter += 1
                    break
                # Check if problem is solved and print best and worst results
                if best_score > 0:
                    print("Problem not solved on generation {} (restarted {} times). Best
solution score is {} and "
                          "worst is {}".format(nb_generations_done, restart_counter, best_score,
worst_score))
                    # Not solved => select a new generation among this ranked population
                    # Retain only the percentage specified by selection rate
                    next_breeders = ga_utils.pick_from_population(ranked_population,
self._selection_rate,
                                                                  self._random_selection_rate)

                    children = ga_utils.create_children_random_parents(next_breeders,
self._nb_children)
                    new_population = ga_utils.mutate_population(children, self._mutation_rate)

                    nb_generations_done += 1
                    overall_nb_generations_done += 1
                else:
                    print("Problem solved after {} generations ({} overall generations)!!!
Solution found is:".
                          format(nb_generations_done, overall_nb_generations_done))
```

```
                    best_solution.display()
                    found = True
                    print("It took {} to solve
it".format(tools.get_human_readable_time(self._start_time, time())))

        if not found:
            print("Problem not solved after {} generations. Printing best and worst results
below".
                  format(overall_nb_generations_done))
            ranked_population = ga_utils.rank_population(new_population)
            best_solution = ranked_population[0]
            worst_solution = ranked_population[-1]
            print("Best is:")
            best_solution.display()
            print("Worst is:")
            worst_solution.display()

    def _load(self):

        if ((self._selection_rate + self._random_selection_rate) / 2) * self._nb_children != 1:
            raise Exception("Either the selection rate, random selection rate or the number of
children is not "
                            "well adapted to fit the population")

        values_to_set = fileloader.load_file_as_values(self._model_to_solve)
        zeros_to_count = '0' if len(values_to_set) < 82 else '00'


        self._start_time = time()
        s = Sudoku(values_to_set)

        return s
```

Class: csp.py

```
class CSP:

    def __init__(self, variables, domains, neighbors, constraints):
        """Construct a CSP problem. If variables is empty, it becomes domains.keys()."""
        variables = variables or list(domains.keys())
        self.variables = variables
        self.domains = domains
        self.neighbors = neighbors
        self.constraints = constraints
        self.initial = ()
        self.curr_domains = None
        self.nassigns = 0
        self.n_bt = 0
def backtracking_search(csp,select_unassigned_variable,order_domain_values,inference):
    def backtrack(assignment):
        if len(assignment) == len(csp.variables):
            return assignment
        var = select_unassigned_variable(assignment, csp)
        for value in order_domain_values(var, assignment, csp):
            if 0 == csp.nconflicts(var, value, assignment):
                csp.assign(var, value, assignment)
                removals = csp.suppose(var, value)
                if inference(csp, var, value, assignment, removals):
                    result = backtrack(assignment)
                    if result is not None:
                        return result
                    else:
                        csp.n_bt += 1
                csp.restore(removals)
        csp.unassign(var, assignment)
        return None

    result = backtrack({})
```

```
    assert result is None or csp.goal_test(result)
    return result
```

Algorithm Analysis:

■ Time complexity

For genetics algorithm,

The time complexity to solve a Sudoku puzzle will depend on a number of factors, including the size of the population, the number of generations, and the specific fitness function and genetic operators used. In general, the time complexity of a genetic algorithm to solve a Sudoku puzzle will be O(n^2 * g), where n is the size of the population and g is the number of generations. This means that as the population size or the number of generations increases, the time complexity of the genetic algorithm will also increase.

For CSP algorithm,

In general, the time complexity of a CSP to solve a Sudoku puzzle will be O(b^m), where b is the average branching factor of the search tree and m is the depth of the tree. In the case of a Sudoku puzzle, the branching factor is typically 9 (since there are 9 possible values that can be placed in each cell), and the depth of the tree is determined by the size of the puzzle. For a standard 9x9 Sudoku puzzle, the depth of the tree will be 81 (since there are 81 cells in the puzzle). This means that the time complexity of a CSP to solve a 9x9 Sudoku puzzle would be O(9^81).

■ Space complexity

For genetics algorithm ,

The space complexity of a genetic algorithm to solve a sudoku puzzle would be O(P * S), where P is the size of the population and S is the size of each individual (in this case, the size of the sudoku puzzle).

For CSP algorithm,

Overall, the space complexity of a constraint satisfaction algorithm to solve a sudoku puzzle would be O(B^D), where B is the branching factor and D is the depth of the search tree. In the case of a sudoku puzzle, the depth of the search tree is equal to the number of variables (81 for a standard 9x9 puzzle) and the branching factor is 9, so the space complexity would be O(9^81). This means that the algorithm would require a large amount of memory to solve the puzzle

■ Completeness

For genetics algorithm,

Overall, genetic algorithms are not considered to be complete algorithms, meaning that they are not guaranteed to find the optimal solution to a problem in all cases.

For CSP algorithm,

In terms of efficiency CSP is not considered to be full "complete" but in the sense of finding solution

CSP is "complete" .

■ Optimality

For genetics algorithm,

Genetic algorithm is not guaranteed to find the optimal solution to a problem in all cases. Therefore, it's not optimal.

For CSP algorithm,

In the same manner CSP is optimal.

Conclusion:

We have used both genetic and CSP algorithm to solve sudoku and it turns out CSP algorithm is more efficient than genetic algorithm , Because CSP ensures an optimal solution where genetic algorithm might fail because of less and random generations where as CSP is a intelligent backtracking algorithm which check all the possible combinations of numbers.