# Deliverable 02: Genetic Algorithms and CSPs

AI-4007: Applied Artificial Intelligence

Submitted by: Sara Qayyum | 20I-0556 | BSSE-6R

Submitted on: March 24, 2023

Student Signatures: Sara

## Introduction

Within the course of this deliverable, the task assigned followed a problem solution pair of utilizing a genetic algorithm for the optimization of scheduling examinations for N courses, in K halls with T timeslots. Additionally, the conflicting student pairs were assessed to develop a workable and optimal schedule. The general outline for the genetic algorithm being followed in my solution followed the below sequence:

a. A population of n random solutions, referred to in the code as ‘individuals’ or ‘chromosomes’, was generated – the aforementioned being an encoded representation

b. Each individual was assigned a fitness score to evaluate it through a fitness function with its basis of penalties of:

   i. Double booking a hall for two or more courses
   ii. Overbooking halls beyond their specified hall hours
   iii. Scheduling courses in one timeslot with conflicting students

c. Based on the initial population, the solutions are passed through an evolutionary loop to achieve:

   i. Selecting fittest solutions from the population using tournament selection
   ii. Reproducing using these as parent solutions with single-point crossovers
   iii. Mutating with a 0.1 mutation rate to exit local maxima, if any

## Deliberating upon Solution

Using the above outline, the below function randomly generates integers within a range of specified halls and timeslots to create an encoded solution, representative of ‘course-timeslot-hall’, e.g., Course 1, timeslot 1, hall 1 would be represented by the string ‘111’.

```
def generate_one_solution(self, courses, hall, timeslots):  # Generating on
potential solution in the population → Therein, referred to as a chromosome
    chromosome = []
    for course in range(courses):
        random_hall = random.randint(1, hall)
        random_slot = random.randint(1, timeslots)
        chromosome.append(str(course + 1) + str(random_slot) + str(random_hall))
    return chromosome
```

The above solutions are generated for a specified number designated to the size of the initial population. This finalizes the initialization of our initial population.

```
def generate_population(self, courses, hall, timeslots, initial_population):  # Generating an entire population of solutions based on the parameter sze of initial_population
    entire_population = []
    for solution in range(initial_population):
        entire_population.append(self.generate_one_solution(courses, hall, timeslots))
    return entire_population
```

This population is fed into a fitness function that works based on assigning a score using penalties as deliberated earlier – a penalty of 10 for double booking, a penalty of 100 for overbooking halls past their exam hours, and a penalty of 1000 for not assessing conflicts into the solution. This is fed into a list called ‘fitness’ that matches the same indexes as the population list.

```
def fitness_function(self, chromosome, conflicts, hall_hours, exam_hours, fitness):  # Creating a fitness function based on penalties of double booking, overbooking and conflicts
    penalty = 0

    # Penalty addition for double booking
    checked = []
    for i in range(len(chromosome)):
        for j in range(i + 1, len(chromosome)):
            if chromosome[i][1:3] == chromosome[j][1:3]:
                if i != j:
                    if chromosome[i][1:3] not in checked:
                        penalty += 10
        checked.append(chromosome[i][1:3])

    # Penalty addition for overbooking
    overbooked = math.floor(hall_hours / exam_hours)
    for i in range(len(chromosome)):
        if int(chromosome[i][1]) > overbooked:
            penalty += 100

    # Penalty addition for conflicting students in the same timeslot
    checked = []
    for i in range(len(chromosome)):
        for j in range(i + 1, len(chromosome)):
            if chromosome[i][1] == chromosome[j][1]:
                for k in range(len(conflicts)):
                    # print(int(conflicts[k][0]), int(chromosome[i][0]), int(conflicts[k][1]), int(chromosome[j][0]))
                    if int(conflicts[k][0]) == int(chromosome[i][0]):
                        if int(conflicts[k][1]) == int(chromosome[j][0]):
                            penalty += 1000
                    if int(conflicts[k][0]) == int(chromosome[j][0]):
                        if int(conflicts[k][1]) == int(chromosome[i][0]):
                            penalty += 1000

    fitness.append(penalty)
```

Using sequential tournament selection of 5 solutions per selection, the fittest of the 5 is re-appended to a cleared-out population.

```
def selection(self, population, fitness):  # Selection criterion based on a
sequential tournament selection methodology
    # Using tournament selection -> Set static at 5
    tournament_size = 5
    selected = []
    for f in range(0, len(population), tournament_size):
        index_of_selected = fitness.index(min(fitness[f:f + 5]))
        selected.append(population[index_of_selected])
    return selected
```

Using the chosen population of this generation brought about in the selection, reproduction is done through crossover with a crossover point of the second character – maintaining the first as static to keep an entire solution consistent with the specified courses.

```
def crossover(self, chromosome_one, chromosome_two, courses):  # Single-point
cross over method applied with a cross-over point of [1]
    offsprings = []
    offsprings_one = []
    offsprings_two = []
    for i in range(courses):
        offspring_one = chromosome_one[i][0] + chromosome_one[i][1] +
chromosome_two[i][2]
        offspring_two = chromosome_two[i][0] + chromosome_two[i][1] +
chromosome_one[i][2]
        offsprings_one.append(offspring_one)
        offsprings_two.append(offspring_two)
    offsprings.append(offsprings_one)
    offsprings.append(offsprings_two)
    return offsprings_one
```

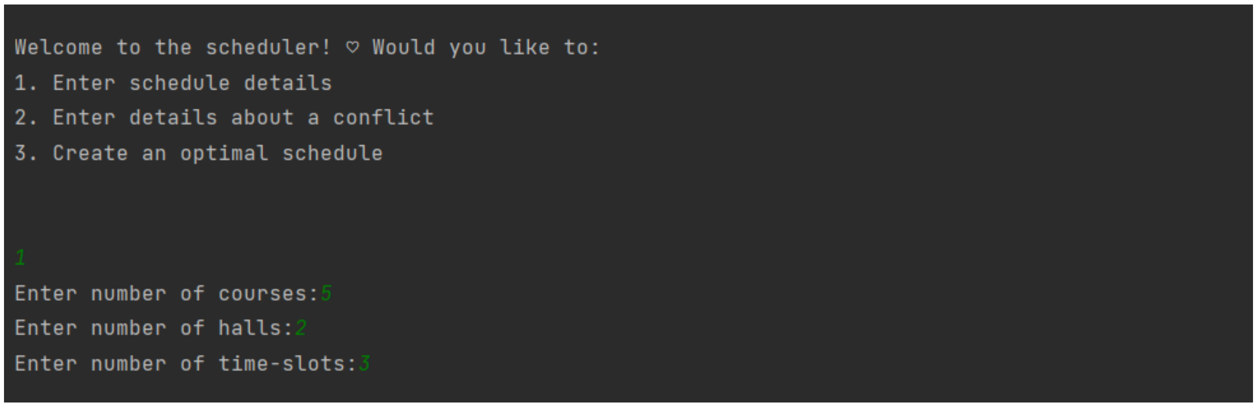
Mutation is brought about with 0.1 probability to avoid local maxima in the evolutionary loop.

```
def mutate(self, mutation_rate, population):  # Mutating solution with a
probability of 0.1
    for individual in population:
        for chromosome in individual:
            if random.random() < mutation_rate:
                temp = str(chromosome[0]) + str(chromosome[2]) +
str(chromosome[1])
                chromosome = temp
```

The driver function ties the above functions together in its sequential order – introducing the evolutionary loop for the first time, that indicates the number of generations for the algorithm.

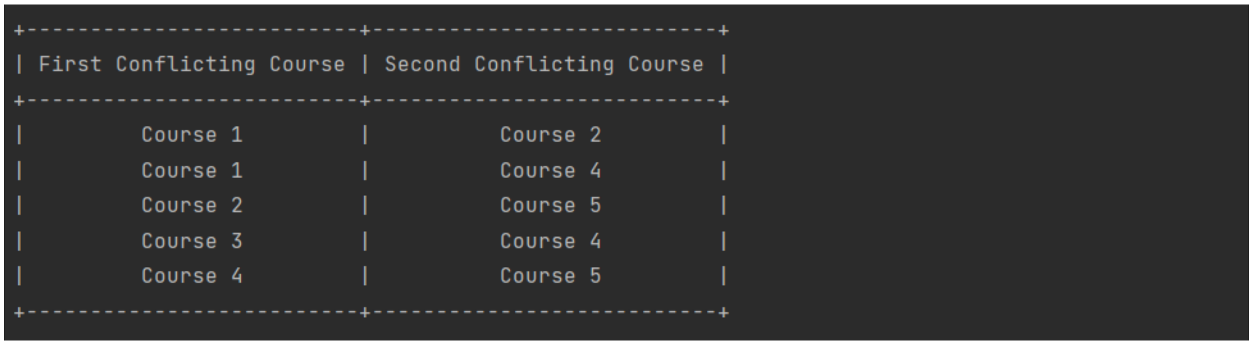
```
def genetic_algorithm_driver(self, courses, halls, timeslots, conflicts,
exam_hours, hall_hours, initial_population,
                             evolutionary_loop_size):
    population = self.generate_population(courses, halls, timeslots,
initial_population)
    fitness = []
    for i in range(len(population)):
        self.fitness_function(population[i], conflicts, hall_hours, exam_hours,
fitness)
    # print(fitness)
    for generation in range(evolutionary_loop_size):
        selected = self.selection(population, fitness)
        population.clear()
        population.extend(selected)
        offsprings = []
        for reproduced in range(1, len(population)):
            for solution in range(courses):
                offspring = self.crossover(population[reproduced - 1],
population[reproduced], courses)
                offsprings.append(offspring)
        population.extend(offsprings)
        population = population[:initial_population]
        self.mutate(0.1, population)
        fitness.clear()
        for fit in range(len(population)):
            self.fitness_function(population[fit], conflicts, hall_hours,
exam_hours, fitness)

    possible_solution_index = fitness.index(min(fitness))
    self.printSolution(population[possible_solution_index], min(fitness))
```

## Example Solutions | Running the algorithm



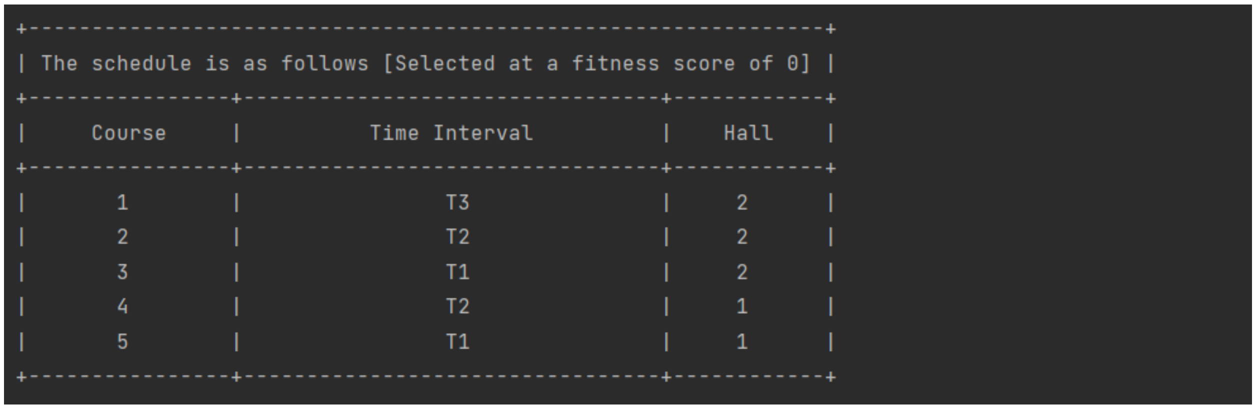
```
Welcome to the scheduler! ♡ Would you like to:
1. Enter schedule details
2. Enter details about a conflict
3. Create an optimal schedule



1
Enter number of courses:5
Enter number of halls:2
Enter number of time-slots:3
```

A console-based, user driven menu that requires specifications of courses, halls, and timeslots



```
+-------------------------+--------------------------+
| First Conflicting Course | Second Conflicting Course |
+-------------------------+--------------------------+
|         Course 1        |          Course 2        |
|         Course 1        |          Course 4        |
|         Course 2        |          Course 5        |
|         Course 3        |          Course 4        |
|         Course 4        |          Course 5        |
+-------------------------+--------------------------+
```

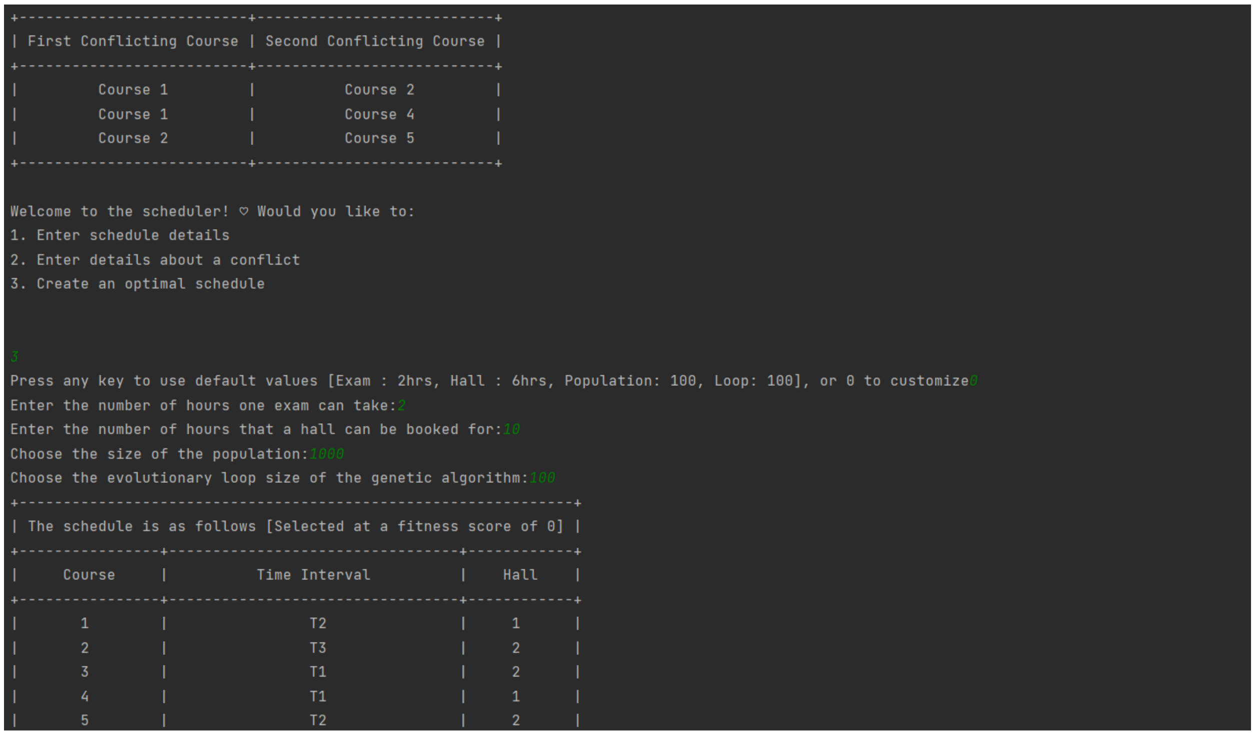
Using the same menu, the details of conflicting courses is added as per the problem in the assignment



```
+-------------------------------------------------------------+
| The schedule is as follows [Selected at a fitness score of 0] |
+----------------+--------------------------------+------------+
|     Course     |          Time Interval         |    Hall    |
+----------------+--------------------------------+------------+
|       1        |               T3               |     2      |
|       2        |               T2               |     2      |
|       3        |               T1               |     2      |
|       4        |               T2               |     1      |
|       5        |               T1               |     1      |
+----------------+--------------------------------+------------+
```

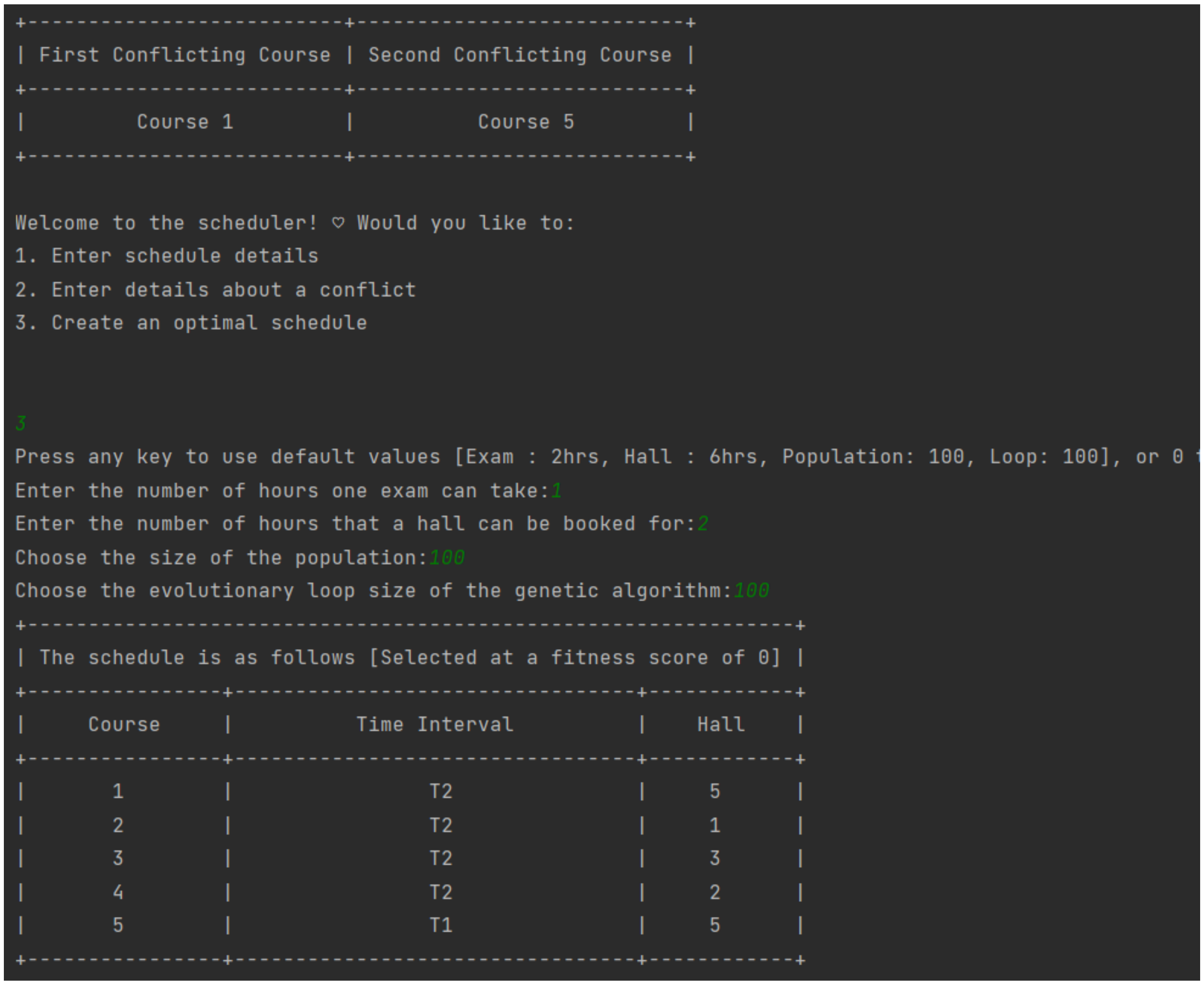
The final output, one of potentially many solutions with a fitness score of 0, with the default parameters [Exam : 2hrs, Hall : 6hrs, Population: 100, Loop: 100]

In the next example, I illustrate that for 5 courses, 2 halls, and 3 timeslots – I catered my solution to include changing exam hall hours, timeslot hours, initial population size, number of generations:



```
+-------------------------+--------------------------+
| First Conflicting Course | Second Conflicting Course |
+-------------------------+--------------------------+
|         Course 1        |          Course 2        |
|         Course 1        |          Course 4        |
|         Course 2        |          Course 5        |
+-------------------------+--------------------------+

Welcome to the scheduler! ♡ Would you like to:
1. Enter schedule details
2. Enter details about a conflict
3. Create an optimal schedule


3
Press any key to use default values [Exam : 2hrs, Hall : 6hrs, Population: 100, Loop: 100], or 0 to customize0
Enter the number of hours one exam can take:2
Enter the number of hours that a hall can be booked for:10
Choose the size of the population:1000
Choose the evolutionary loop size of the genetic algorithm:100
+-------------------------------------------------------------+
| The schedule is as follows [Selected at a fitness score of 0] |
+----------------+--------------------------------+------------+
|     Course     |          Time Interval         |    Hall    |
+----------------+--------------------------------+------------+
|       1        |               T2               |     1      |
|       2        |               T3               |     2      |
|       3        |               T1               |     2      |
|       4        |               T1               |     1      |
|       5        |               T2               |     2      |
```

In the next example, I illustrate that for 5 courses, 5 halls, and 5 timeslots – I lessen the conflicts, decrease the timeslot duration and the hall hours:



```
+-------------------------+-------------------------+
| First Conflicting Course | Second Conflicting Course |
+-------------------------+-------------------------+
|         Course 1         |          Course 5         |
+-------------------------+-------------------------+

Welcome to the scheduler! ♡ Would you like to:
1. Enter schedule details
2. Enter details about a conflict
3. Create an optimal schedule


3
Press any key to use default values [Exam : 2hrs, Hall : 6hrs, Population: 100, Loop: 100], or 0
Enter the number of hours one exam can take:1
Enter the number of hours that a hall can be booked for:2
Choose the size of the population:100
Choose the evolutionary loop size of the genetic algorithm:100
+------------------------------------------------------------+
| The schedule is as follows [Selected at a fitness score of 0] |
+----------------+--------------------------------+------------+
|     Course     |          Time Interval          |    Hall    |
+----------------+--------------------------------+------------+
|       1        |               T2               |     5      |
|       2        |               T2               |     1      |
|       3        |               T2               |     3      |
|       4        |               T2               |     2      |
|       5        |               T1               |     5      |
+----------------+--------------------------------+------------+
```

The first program, with majority conflicts, ended with– "Thank you for using the algorithm! It took **0.24622321128845215 seconds to run**."

The advantages of using genetic algorithms as the optimization technique for this problem includes the feasibility it provides against its methodology – one can play around with factors of number of generations, mutation rates, crossover rates, population sizes to adjust the algorithm to a specific problem, allowing for it to be robust. Similarly, the sheer number of selection techniques allows the adaptation of the algorithm to this particular problem to best choose selection parameters. However, this comes at a computational cost for simple problems like scheduling with lesser conflicts, with smaller search spaces taking a longer time. Where there is flexibility in selection parameters, novices may not always know which parameters to utilize to achieve best results to a specific problem. Another major disadvantage is that a genetic algorithm does not always guarantee the best solution given the randomness of the population generated.