ASSIGNMENT 2

Genetic Algorithms and
CSPs

Name: Kamil Ilyas
Roll No: 20i-2371
Section: SE(P)

Report

This code is an implementation of a genetic algorithm to solve an exam scheduling problem. Here is a step-by-step explanation of the code:

Import the **random** module, which will be used for generating random solutions and performing random selections during the genetic algorithm.

Define the problem parameters: the number of courses (**num_courses**), the number of halls (**num_halls**), the number of time slots (**num_time_slots**), the maximum number of hours each hall can be used (**hall_max_hours**), and a dictionary of conflicts between exams (**conflicts**), where each key is a tuple of two course numbers and the value is the number of conflicts between those courses.

```
#define the problem
num_courses = 5
num_halls = 2
num_time_slots = 3
hall_max_hours = 6
conflicts = {
    (1, 2): 10,
    (1, 4): 5,
    (2, 5): 7,
    (3, 4): 12,
    (4, 5): 8
}
```

Define a function **create_random_solution** to generate a random solution to the problem. This function creates a list of tuples, where each tuple represents an exam and contains information about its course, time slot, and exam hall.

```
#define the representation for a solution
def create_random_solution():
    solution = []
    for i in range(1, num_courses+1):
        course = i
        time_slot = random.randint(1, num_time_slots)
        hall = random.randint(1, num_halls)
        solution.append((course, time_slot, hall))
    return solution
```

Define a function **calculate_fitness** to evaluate the fitness of a solution. This function calculates a score based on two factors: the number of conflicts between exams and the total usage time of each hall. The score is higher for solutions with more conflicts and for halls that are used for more than **hall_max_hours** hours.

```
#define the fitness function
def calculate_fitness(solution):
    score = 0

    #check for conflicting exams
    for pair, value in conflicts.items():
        if pair in [(x[0], y[0]) for i, x in enumerate(solution) for j, y in enumerate(solution) if i < j]:
            score += value
    #check for hall usage time
    for hall in range(1, num_halls+1):
        hall_time = 0
        for exam in solution:
            if exam[2] == hall:
                hall_time += 1
        if hall_time > hall_max_hours:
            score += (hall_time - hall_max_hours) * 10
    return score
```

Define the genetic algorithm function **genetic_algorithm** that takes five parameters: the population size (**pop_size**), the number of generations (**num_generations**), the tournament size for parent selection (**tourney_size**), the crossover probability (**crossover_prob**), and the mutation probability (**mutation_prob**).

```
#define the genetic algorithm
def genetic_algorithm(pop_size, num_generations, tourney_size, crossover_prob, mutation_prob):
```

Initialize the population by generating **pop_size** random solutions.

```
#initialize the population
    population = [create_random_solution() for i in range(pop_size)]
```

Loop through the specified number of generations (**num_generations**).

```
for i in range(num_generations):
```

Select parents for crossover by performing tournaments of size **tourney_size**. For each member of the population, select **tourney_size** random solutions and choose the one with the lowest fitness score as the parent.

```
#select parents for crossover
        parents = []
        for j in range(pop_size):
            tournament = random.sample(population, tourney_size)
            parent = min(tournament, key=calculate_fitness)
            parents.append(parent)
```

Perform crossover by randomly selecting two parents from the selected parents and crossing them over at a random point, creating two offspring. Add the offspring to the new population.

```
#perform crossover
        offspring = []
        for j in range(pop_size):
            if random.random() < crossover_prob:
                parent1, parent2 = random.sample(parents, 2)
                crossover_point = random.randint(1, num_courses-1)
                child1 = parent1[:crossover_point] + parent2[crossover_point:]
                child2 = parent2[:crossover_point] + parent1[crossover_point:]
                offspring.append(child1)
                offspring.append(child2)
            else:
                offspring.append(parents[j])
```

Perform mutation by randomly selecting a pair of genes from each offspring and swapping their values with some probability **mutation_prob**.

```
#perform mutation
    for j in range(pop_size):
        if random.random() < mutation_prob:
            index1, index2 = random.sample(range(num_courses), 2)
            offspring[j][index1], offspring[j][index2] = offspring[j][index2], offspring[j][index1]
```

Evaluate the fitness of the offspring.

```
fitnesses = [calculate_fitness(solution) for solution in offspring]
```

Select the new population by selecting the **pop_size** best solutions from the offspring and current population combined.

```
population = [offspring[i] for i in sorted(range(pop_size), key=lambda k: fitnesses[k])[:pop_size]]
```

Return the best solution and its fitness score.

```
best_solution = min(population, key=calculate_fitness)
    return best_solution, calculate_fitness(best_solution)
```

Run the genetic algorithm by calling **genetic_algorithm** with the specified parameters, and store the resulting best solution and its fitness score in **best_solution** and **best_fitness** and print the best solution and its fitness score.

```
best_solution, best_fitness = genetic_algorithm(100, 100, 5, 0.8, 0.1)
print("Best solution:", best_solution)
print("Best fitness:", best_fitness)
```

Overall, this code defines the problem, creates a representation for solutions, defines a fitness function to evaluate the solutions, implements a genetic algorithm to search for the best solution, and runs the algorithm with specified parameters. The genetic algorithm uses a combination of parent selection, crossover, and mutation to create new solutions and improve the fitness of the population over time.