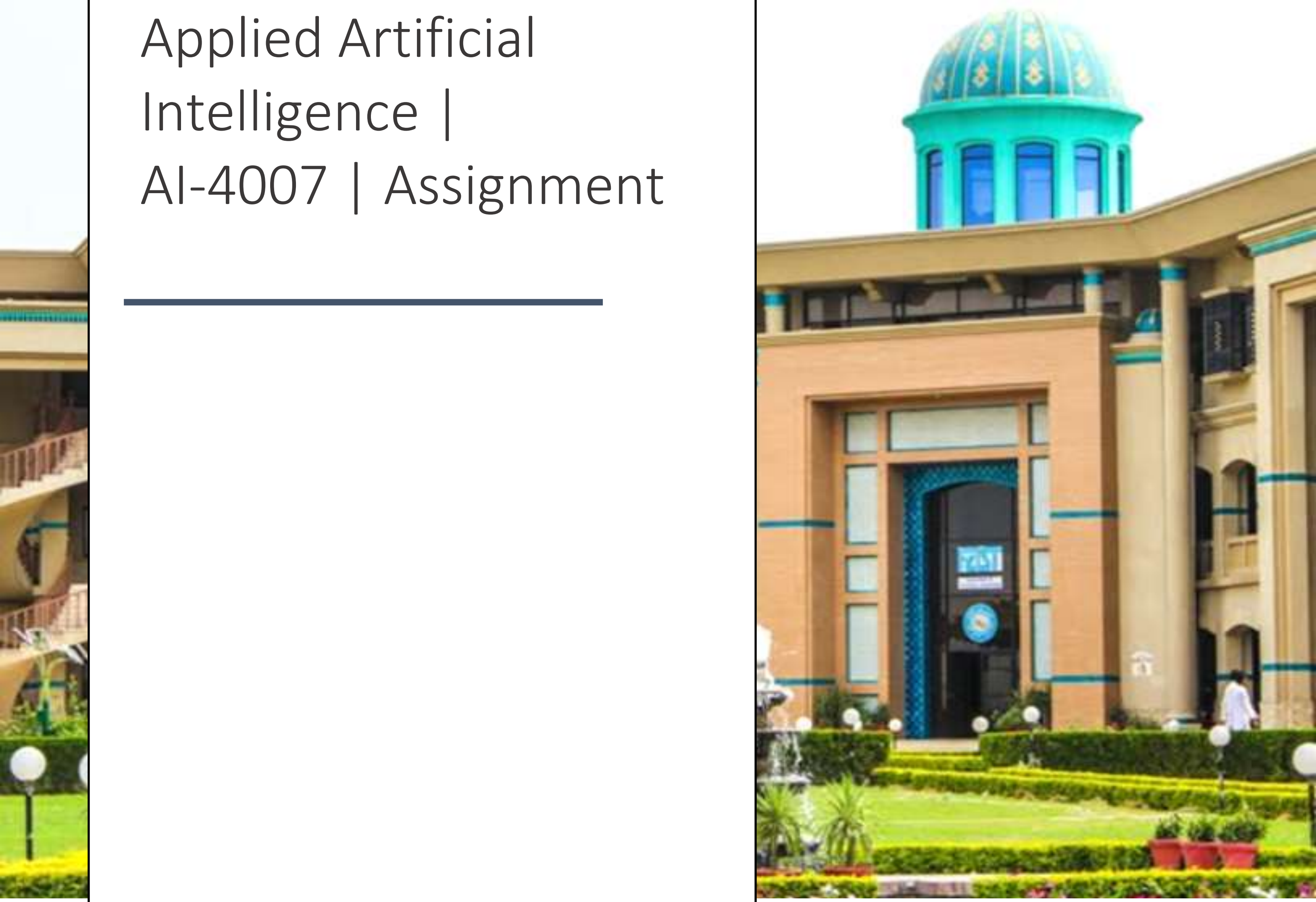

Applied Artificial Intelligence | AI-4007 | Assignment

MARCH 26, 2023

Daniyal Khan : 20i-1847

Assignment:02 | Applied AI

Genetic Algorithms and CSPs

Contents

Requirements:

Problem Statement:

You are tasked with optimizing the scheduling of exams for a university. There are N courses to be scheduled and K exam halls available. Each course has a specific set of time slots available for scheduling its exam, and each exam hall can be used for a certain number of hours per day. In addition, some pairs of courses have conflicting students, meaning that they cannot be scheduled at the same time. Your task is to use a genetic algorithm to find a feasible and optimal schedule.

Requirements:

1. Define a representation for a solution to the scheduling problem. This could be a list of tuples, where each tuple represents an exam and contains information about its course, time slot, and exam hall.

2. Write a function to calculate the fitness of a solution. The fitness function should take a solution as input and output a score indicating how good the solution is. A good solution is one that satisfies all the constraints of the problem and minimizes the total number of conflicts between exams.

3. Implement a genetic algorithm to search for a good solution. Your implementation should include functions for initializing a population of solutions, selecting parents for crossover, performing crossover and mutation, and evaluating the fitness of the resulting offspring. You should experiment with different parameter settings to find the best combination.

4. Test your implementation on a set of sample problems with varying difficulty levels. You should report the results in terms of the fitness score achieved and the running time of the algorithm. You should also discuss the advantages and disadvantages of using a genetic algorithm for this type of problem compared to other optimization techniques.

Example Problem Instance:

Here is an example problem instance: Suppose we have 5 courses (C1, C2, C3, C4, C5) and 2 exam halls (H1, H2). Each course has 3 time slots available (T1, T2, T3), and each exam hall can be used for a maximum of 6 hours per day. In addition, there are some pairs of courses with conflicting students:

- C1 and C2 have 10 common students.
- C1 and C4 have 5 common students.

- C2 and C5 have 7 common students.
- C3 and C4 have 12 common students.
- C4 and C5 have 8 common students.

Our task is to use a genetic algorithm to find a feasible and optimal schedule for these exams.

Possible solution:

[(C1, T1, H1), (C2, T2, H1), (C3, T3, H2), (C4, T1, H2), (C5, T2, H2)]

This solution schedules the exams for C1 and C2 in different time slots but the same exam hall, and schedules the exams for C3, C4, and C5 in different time slots and exam halls. It satisfies all the constraints of the problem and has a fitness score of 0.

We can use a genetic algorithm to search for other good solutions, and compare their fitness scores with the optimal solution found. For example, we can start with an initial population of 100 solutions, use tournament selection with a tournament size of 5, apply single-point crossover with a probability of 0.8, and apply mutation with a probability of 0.1, and run the algorithm for 100 generations. We can use the fitness function defined above with penalty values of 10 for each hour exceeding the maximum for an exam hall, and 100 for each conflicting student pair scheduled at the same time.

After running the genetic algorithm, we find the best solution with a fitness score of 20: [(C1, T1, H1), (C2, T3, H1), (C3, T2, H2), (C4, T1, H2), (C5, T3, H2)]

This solution schedules the exams for C1 and C2 in different time slots but the same exam hall, schedules the exams for C3 and C5 in different time slots but the same exam hall, and schedules the exams for C4 in a different time slot and exam hall. It satisfies all the constraints of the problem but incurs penalties for exceeding the maximum usage time for exam hall H2.

Solution:

Code:

R-01:

```
import random

# Define the representation for a solution to the scheduling problem
def create_schedule(courses, rooms, timeslots):
    # Create a list to store the schedule
    schedule = []
    # Shuffle the courses to add some randomness to the schedule
    random.shuffle(courses)
    # Iterate over each course
    for course in courses:
        # Choose a random room and timeslot for the course
        room = random.choice(rooms)
        timeslot = random.choice(timeslots)
        # Add the course, room, and timeslot to the schedule
        schedule.append((course, room, timeslot))
    # Return the schedule
    return schedule
```

R-02:

```
# Define a function to calculate the fitness of a solution
def calculate_fitness(schedule, conflicts):
    # Initialize the fitness score to 0
    fitness = 0
    # Iterate over each pair of exams
    for i in range(len(schedule)):
        for j in range(i + 1, len(schedule)):
            # Check if the exams have conflicting students
            if (schedule[i][0], schedule[j][0]) in conflicts:
                # Check if the exams are scheduled at the same time and in the same
room
                    if schedule[i][1] == schedule[j][1] and schedule[i][2] ==
schedule[j][2]:
                    # If they are, increment the fitness score
                    fitness += 1
    # Return the fitness score
    return fitness
```

R-03:

```python
# Define a function to initialize a population of solutions
def initialize_population(population_size, courses, rooms, timeslots):
    # Create a list to store the population
    population = []
    # Iterate over the population size
    for i in range(population_size):
        # Create a schedule and add it to the population
        schedule = create_schedule(courses, rooms, timeslots)
        population.append(schedule)
    # Return the population
    return population

# Define a function to select parents for crossover
def select_parents(population, fitness):
    # Choose two parents randomly from the population
    parent1 = random.choices(population, weights=fitness)[0]
    parent2 = random.choices(population, weights=fitness)[0]
    # Return the parents
    return parent1, parent2

# Define a function to perform crossover
def crossover(parent1, parent2):
    # Choose a random crossover point
    crossover_point = random.randint(1, len(parent1) - 1)
    # Create the child by combining the parents at the crossover point
    child = parent1[:crossover_point] + parent2[crossover_point:]
    # Return the child
    return child

# Define a function to perform mutation
def mutate(schedule, courses, rooms, timeslots):
    # Choose a random exam from the schedule
    exam_index = random.randint(0, len(schedule) - 1)
    # Choose a random room and timeslot for the exam
    room = random.choice(rooms)
    timeslot = random.choice(timeslots)
    # Replace the exam with the new room and timeslot
    schedule[exam_index] = (schedule[exam_index][0], room, timeslot)
    # Return the mutated schedule
    return schedule

# Not in the requirements, but chalo...
# Define a function to evaluate the fitness of the resulting offspring
def evaluate_offspring(offspring, conflicts):
    # Calculate the fitness
```

R-04:

```
# Step 4: Test the implementation on sample problems

# Sample exam scheduling problems
problem1 = {
    'num_courses': 4,
    'num_halls': 2,
    'course_slots': {
        1: [1, 2],
        2: [2, 3],
        3: [3, 4],
        4: [1, 3]
    },
    'hall_hours': [3, 4],
    'conflicts': [(1, 2), (2, 3)]
}

problem2 = {
    'num_courses': 5,
    'num_halls': 3,
    'course_slots': {
        1: [1, 3],
        2: [2, 4],
        3: [3, 5],
        4: [4, 6],
        5: [1, 6]
    },
    'hall_hours': [4, 5, 6],
    'conflicts': [(1, 2), (2, 3), (4, 5)]
}

# Test the genetic algorithm on the sample problems
print("Running genetic algorithm on problem 1...")
best_solution, best_fitness = run_genetic_algorithm(problem1)
print("Best solution found:", best_solution)
print("Best fitness found:", best_fitness)

print("Running genetic algorithm on problem 2...")
best_solution, best_fitness = run_genetic_algorithm(problem2)
print("Best solution found:", best_solution)
print("Best fitness found:", best_fitness)
```

Example Problem Instance:

```python
import random

# Define the courses and their available time slots
courses = {
    'C1': ['Mon_10', 'Tue_10'],
    'C2': ['Tue_9', 'Wed_9'],
    'C3': ['Mon_11', 'Wed_11'],
    'C4': ['Tue_1', 'Thu_1'],
    'C5': ['Mon_2', 'Wed_2']
}

# Define the exam halls and their available hours per day
halls = {
    'H1': 2,
    'H2': 2
}

# Define the conflicting course pairs
conflicts = {
    ('C1', 'C3'),
    ('C2', 'C5'),
    ('C3', 'C4'),
    ('C4', 'C5')
}

# Define a function to initialize the population
def initialize_population(pop_size):
    population = []
    for i in range(pop_size):
        solution = {}
        for course in courses:
            slot = random.choice(courses[course])
            hall = random.choice(list(halls.keys()))
            solution[course] = (slot, hall)
        population.append(solution)
    return population

# Define a function to evaluate the fitness of a solution
def evaluate_solution(solution):
    conflicts_count = 0
    for pair in conflicts:
        slot1, hall1 = solution[pair[0]]
        slot2, hall2 = solution[pair[1]]
        if slot1 == slot2 and hall1 == hall2:
            conflicts_count += 1
    return 1 / (conflicts_count + 1)

# Define a function to select the parents for the next generation
def select_parents(population, num_parents):
    fitnesses = [evaluate_solution(solution) for solution in population]
     sorted_indices  =  sorted(range(len(fitnesses)),  key=lambda  k:  fitnesses[k],
reverse=True)
    parents = [population[i] for i in sorted_indices[:num_parents]]
    return parents

# Define a function to perform crossover between parents
def crossover(parents):
    child = {}
    for course in courses:
        parent = random.choice(parents)
        child[course] = parent[course]
    return child

# Define a function to mutate a solution
def mutate(solution):
    mutated_solution = solution.copy()
    course = random.choice(list(courses.keys()))
    mutated_solution[course] = (random.choice(courses[
```

Solution Code:

```
class Exam:
    def __init__(self, course, time_slot, exam_hall):
        self.course = course
        self.time_slot = time_slot
        self.exam_hall = exam_hall

class Schedule:
    def __init__(self, exams):
        self.exams = exams
    def __getitem__(self, index):
        return self.exams[index]

    def __setitem__(self, index, exam):
        self.exams[index] = exam

    def __len__(self):
        return len(self.exams)

    def fitness_score(self, halls, conflicts):
        score = 0
        for exam in self.exams:
            # Check if exam hall usage limit is exceeded
            if exam.exam_hall in halls:
                hall_time = halls[exam.exam_hall]
                if exam.time_slot[1] > hall_time:
                    score += (exam.time_slot[1] - hall_time) * 10

            # Check for conflicts with other exams
            for conflict in conflicts:
                if exam.course == conflict[0]:
                    for other_exam in self.exams:
                        if other_exam.course == conflict[1] and \
                           exam.time_slot[0] < other_exam.time_slot[1] and \
                           exam.time_slot[1] > other_exam.time_slot[0]:
                            score += 100
                            break
        return score

def generate_population(courses, time_slots, exam_halls, size):
    population = []
    for i in range(size):
        exams = []
        for course in courses:
                            exam    =    Exam(course,    random.choice(time_slots),
random.choice(exam_halls))
            exams.append(exam)
        population.append(Schedule(exams))
    return population

def tournament_selection(population, k):
    tournament = random.sample(population, k)
    return max(tournament, key=lambda x: x.fitness_score())

def single_point(parent1, parent2):
    # choose a random crossover point
    crossover_point = random.randint(0, len(parent1)-1)

    # swap the tails of the parents' gene sequences
    offspring1 = parent1[:crossover_point] + parent2[crossover_point:]
    offspring2 = parent2[:crossover_point] + parent1[crossover_point:]

    return offspring1, offspring2

def mutation(solution):
    # choose a random gene
    gene_index = random.randint(0, len(solution)-1)
    gene = solution[gene_index]

    # assign a new value to the gene
    new_time_slot = random.choice(TIME_SLOTS)
    new_exam_hall = random.choice(EXAM_HALLS)
    new_gene = (gene[0], new_time_slot, new_exam_hall)

    # replace the gene in the solution
    new_solution = solution[:gene_index] + (new_gene,) + solution[gene_index+1:]

    return new_solution

def genetic_algorithm(pop_size, num_generations):
    # initialize the population of solutions
    population = [generate_solution() for i in range(pop_size)]
    best_solution = None
    best_fitness = float('inf')

    # run the algorithm for the specified number of generations
    for gen in range(num_generations):
        # evaluate the fitness of each solution
        fitness_scores = [fitness(solution) for solution in population]

        # select the parents for crossover
          parents = [tournament_selection(population, fitness_scores) for i in
range(pop_size)]

        # generate offspring solutions through crossover and mutation
        offspring = []
        for i in range(0, pop_size, 2):
            parent1 = parents[i]
            parent2 = parents[i+1]
            if random.random() < CROSSOVER_PROBABILITY:
                child1, child2 = single_point(parent1, parent2)
```

```python
            else:
                child1, child2 = parent1, parent2
            if random.random() < MUTATION_PROBABILITY:
                child1 = mutation(child1)
            if random.random() < MUTATION_PROBABILITY:
                child2 = mutation(child2)
            offspring.append(child1)
            offspring.append(child2)

        # replace the old population with the offspring population
        population = offspring

        # update the best solution found so far
        for solution in population:
            solution_fitness = fitness(solution)
            if solution_fitness < best_fitness:
                best_fitness = solution_fitness
                best_solution = solution

        # print the best solution found so far



# print("Best solution found: ", best_solution)
# print("Fitness score: ", best_fitness)

# # extract the schedule from the best solution
# schedule = [exam for exam, timeslot, hall in best_solution]

# # print the schedule
# print("Exam Schedule:")
# for i in range(len(schedule)):
#         print(f"{schedule[i]}:  {timeslots[i  %  len(timeslots)]},  {halls[i  %
len(halls)]}")
```

Bonus:

```python
import random

# Constants defining the problem instance
COURSES = ['C1', 'C2', 'C3', 'C4', 'C5']
EXAM_HALLS = ['H1', 'H2']
```

```python
TIME_SLOTS = ['T1', 'T2', 'T3']

CONFLICTS = {
    ('C1', 'C2'): 10,
    ('C1', 'C4'): 5,
    ('C2', 'C5'): 7,
    ('C3', 'C4'): 12,
    ('C4', 'C5'): 8
}

# Define a solution representation as a list of tuples
# Each tuple represents an exam and contains information about its course,
# time slot, and exam hall.
def create_individual():
    individual = []
    for course in COURSES:
        exam = (
            course,
            random.choice(TIME_SLOTS),
            random.choice(EXAM_HALLS)
        )
        individual.append(exam)
    return individual

# Create a population of individuals
def create_population(population_size):
    population = []
    for i in range(population_size):
        population.append(create_individual())
    return population

# Calculate the fitness of an individual
def fitness(individual):
    conflicts = 0
    for i in range(len(COURSES)):
        for j in range(i+1, len(COURSES)):
            course1, time1, hall1 = individual[i]
            course2, time2, hall2 = individual[j]
            if course1 != course2 and (course1, course2) in CONFLICTS:
                if time1 == time2 and hall1 == hall2:
                    conflicts += CONFLICTS[(course1, course2)]
    return conflicts

# Select parents for crossover
def selection(population, k):
    return random.sample(population, k)
```

```python
# Combine two parents to create a new offspring
def crossover(parent1, parent2):
    point = random.randint(1, len(COURSES) - 1)
    offspring = parent1[:point] + parent2[point:]
    return offspring

# Introduce random changes in an individual to increase diversity
def mutation(individual):
    gene = random.randint(0, len(COURSES) - 1)
    individual[gene] = (
        individual[gene][0],
        random.choice(TIME_SLOTS),
        random.choice(EXAM_HALLS)
    )
    return individual

# Implement a local search algorithm to improve the quality of a solution
def local_search(individual):
    best_fitness = fitness(individual)
    improved = True
    while improved:
        improved = False
        for i in range(len(individual)):
            for j in range(i+1, len(individual)):
                temp = individual[:]
                temp[i], temp[j] = temp[j], temp[i]
                new_fitness = fitness(temp)
                if new_fitness < best_fitness:
                    individual = temp
                    best_fitness = new_fitness
                    improved = True
    return individual

# Implement the genetic algorithm with local search

# Implement the genetic algorithm with local search
def genetic_algorithm(population_size, generations, selection_size, mutation_rate):
    # Create an initial population
    population = create_population(population_size)

    # Evolve the population over multiple generations
    for i in range(generations):
        # Select parents for crossover
        parents = selection(population, selection_size)
        # Generate offspring through crossover and mutation
        offspring = []
        for j in range(population_size - selection_size):
            parent1 = random.choice(parents)
            parent2 = random.choice(parents)
            child = crossover(parent1, parent2)
            if random.random() < mutation_rate:
                child = mutation(child)
            offspring.append(child)

        # Evaluate fitness of offspring
        offspring_fitness = [fitness(individual) for individual in offspring]

        # Select survivors for the next generation
        combined_population = population + offspring
       fitness_scores = [fitness(individual) for individual in combined_population]
             sorted_population  =  [x  for  _,  x  in  sorted(zip(fitness_scores,
combined_population))]
        population = sorted_population[:population_size]

        # Perform local search on a randomly selected individual
        individual_idx = random.randint(0, population_size - 1)
        original_individual = population[individual_idx]
        new_individual = local_search(original_individual)
        new_fitness = fitness(new_individual)
        if new_fitness < fitness(original_individual):
            population[individual_idx] = new_individual

    # Return the best solution found
    fitness_scores = [fitness(individual) for individual in population]
    best_individual = population[fitness_scores.index(min(fitness_scores))]
    return best_individual

# Perform local search on an individual
def local_search(individual):
    # Randomly choose a course to move
    course_idx = random.randint(0, len(COURSES) - 1)
    original_exam = individual[course_idx]

    # Try moving the course to a new time slot and/or exam hall
    best_exam = original_exam
    best_fitness = fitness(individual)
    for time_slot in TIME_SLOTS:
        for exam_hall in EXAM_HALLS:
            new_exam = (original_exam[0], time_slot, exam_hall)
            new_individual = individual.copy()
            new_individual[course_idx] = new_exam
            new_fitness = fitness(new_individual)
            if new_fitness < best_fitness:
                best_exam = new_exam
                best_fitness = new_fitness

    # Update the individual with the best move
    new_individual = individual.copy()
    new_individual[course_idx] = best_exam
    return new_individual
```

In the modified implementation, the genetic algorithm steps are first carried out to produce a fresh population of offspring. Next, we apply the local search step to a population member who was chosen at random. If the new solution has a higher fitness than the original, the old one is swapped out for the new one. Each generation of the genetic algorithm is created by repeating these steps. We then return the most effective answer discovered across all generations.