ARTIFICIAL INTELLIGENCE

COURSE PROJECT

Deadline: Friday, 10th May, 2024, 11:59PM

Instructions:

1. The project is to be done individually. You must complete this project by yourself. You cannot work with anyone else in the class or with someone outside of the class.
1. You must use python programming language.
2. Plagiarism of any kind (copying from others and copying from the internet, etc.,) is not allowed and can result in zero marks in whole project category.
3. Your code must be properly commented.
4. No marks will be assigned if any of the following deliverables are missing.
    a. The source code of the program.
    b. A pdf or word report containing a brief explanation of the steps involved in the project and the results obtained.
5. Put both source code and report in one folder, ZIP it and submit it. Your folder must be named as ROLLNO_NAME.ZIP.

Late submissions will not be accepted.

Project Description:

Timetable scheduling problem is a classical problem in the university environment where a timetable is to be made individually for each semester ensuring minimum number of clashes between sections, professors, and rooms. Thus, we have a timetabling problem where time slots are assigned to each section in a particular room to be taught by a particular professor to teach a particular course.

Please note that each week, there are 2 classes per theory course (each of 1 hour 20 mins length), and lab courses will have a single long session of 3 hours.

The following constraints need to be followed:

Hard Constraints:

These are the constraints that need to be implemented in your project completely and without any compromises.

- Classes can only be scheduled in free classrooms.
- A classroom should be big enough to accommodate the section. There should be two categories of classrooms: classroom (60) and large hall (120).
- A professor should not be assigned two different lectures at the same time.
- The same section cannot be assigned to two different rooms at the same time.
- A room cannot be assigned for two different sections at the same time.
- No professor can teach more than 3 courses.
- No section can have more than 5 courses in a semester.
- Each course would have two lectures per week not on the same or adjacent days.
- Lab lectures should be conducted in two consecutive slots.

- 15 mins breaks allowed between consecutive classes to ensure that there is sufficient time for transitions between classes.

Soft Constraints:

These are the constraints where some compromise can be expected. Hence, you are expected to implement them as best as possible.

- All the theory classes should be taught in the morning session and all the lab sessions should be done in the afternoon session.
- Teachers/students may be facilitated by minimizing the number of floors they have to traverse. That is, as much as possible, scheduled classes should be on the same floor for either party.
- A class should be held in the same classroom across the whole week.
- Teachers may prefer longer blocks of continuous teaching time to minimize interruptions and maximize productivity except when the courses are different.

Other Details:

- The timetable should cover all the 5 days of the week with the morning session from 8:30 – 2:30 and the afternoon session from 2:30 – 5:30.
- The chromosomes should be binary encoded with the following information:
    - Course, Theory/Lab, Section, Section-Strength, Professor, First-lecture-day, First- lecture-timeslot, First-lecture-room, First-lecture-room-size, Second-lecture-day, Second-lecture-timeslot, Second-lecture-room, Second-lecture-room-size
- The fitness function should be an inverse or negative of the sum of all the conflicts/clashes.

Foundational Code:

Follow the classical genetic algorithms' cycle as given in the book with following steps: reproduction, crossover, and mutation.

```
**********************************************************************
*******

# initial population of random bitstring

pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]


# enumerate generations

 for gen in range(n_iter):


# evaluate all candidates in the population

scores = [objective(c) for c in pop]
```

```
# tournament selection
def selection(pop, scores, k=3):
 # first random selection
 selection_ix = randint(len(pop))
 for ix in randint(0, len(pop), k-1):
 # check if better (e.g. perform a tournament)
 if scores[ix] < scores[selection_ix]:
 selection_ix = ix
 return pop[selection_ix]

# select parents
selected = [selection(pop, scores) for _ in range(n_pop)]

# crossover two parents to create two children
def crossover(p1, p2, r_cross):
 # children are copies of parents by default
 c1, c2 = p1.copy(), p2.copy()
 # check for recombination
 if rand() < r_cross:
 # select crossover point that is not on the end of the string
 pt = randint(1, len(p1)-2)
 # perform crossover
 c1 = p1[:pt] + p2[pt:]
 c2 = p2[:pt] + p1[pt:]
 return [c1, c2]

# mutation operator
def mutation(bitstring, r_mut):
 for i in range(len(bitstring)):
 # check for a mutation
```

```
 if rand() < r_mut:
 # flip the bit
 bitstring[i] = 1 - bitstring[i]



...
# create the next generation
children = list()
for i in range(0, n_pop, 2):
 # get selected parents in pairs
 p1, p2 = selected[i], selected[i+1]
 # crossover and mutation
 for c in crossover(p1, p2, r_cross):
 # mutation
 mutation(c, r_mut)
 # store for next generation
 children.append(c)


# genetic algorithm
def genetic_algorithm(objective, n_bits, n_iter, n_pop, r_cross,
r_mut):
 # initial population of random bitstring
 pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]
 # keep track of best solution
 best, best_eval = 0, objective(pop[0])
 # enumerate generations
 for gen in range(n_iter):
 # evaluate all candidates in the population
 scores = [objective(c) for c in pop]
 # check for new best solution
```

```
 for i in range(n_pop):
 if scores[i] < best_eval:
 best, best_eval = pop[i], scores[i]
 print(">%d, new best f(%s) = %.3f" % (gen,  pop[i], scores[i]))
 # select parents
 selected = [selection(pop, scores) for _ in range(n_pop)]
 # create the next generation
 children = list()
 for i in range(0, n_pop, 2):
 # get selected parents in pairs
 p1, p2 = selected[i], selected[i+1]
 # crossover and mutation
 for c in crossover(p1, p2, r_cross):
 # mutation
 mutation(c, r_mut)
 # store for next generation
 children.append(c)
 # replace population
 pop = children
 return [best, best_eval]
```

****
****