Final Report

AI-2002 Artificial Intelligence

Semester Project

Name

Muneeb Ur Rehman

i210392

Section

G

Submitted to:

Dr Labiba

Department of Computer Science BS(CS)

FAST-NUCES Islamabad

Introduction:

The project aimed to address the complex problem of timetable scheduling in educational institutions using genetic algorithms (GAs). Timetable scheduling involves assigning courses, professors, classrooms, and timeslots while satisfying various constraints. GAs, inspired by natural selection, are powerful optimization techniques suitable for solving combinatorial optimization problems like timetable scheduling.

Methodology:

The project followed these key steps:

- **Chromosome Representation:** The chromosomes are 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
- **Genetic Operators:** Implemented selection, crossover, and mutation operators to evolve the population of schedules.
- **Fitness Function:** Defined a fitness function to evaluate the quality of schedules based on constraints and objectives on the following constraints.

- Hard Constraints:

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

9. Lab lectures should be conducted in two consecutive slots.

- Soft constraints:

1. All the theory classes should be taught in the morning session and all the lab sessions should be done in the afternoon session.
2. A class should be held in the same classroom across the whole week.

The fitness function is the inverse or negative of the sum of all the conflicts/clashes.

Experimental Setup:

Parameters: Configured population size as 10, one point crossover, mutation rate is 0.1, and termination criteria is up to 20 generations.

Results and Analysis:

Performance Analysis: Analyzed the effectiveness of the algorithm in producing high-quality schedules based on the minimum fitness value. The time complexity of the genetic algorithm is as follows:
<u>O(g(nm + nm + n))</u>

Conclusion:

The project successfully demonstrated the application of genetic algorithms to solve the timetable scheduling problem. The results showed that GAs could produce schedules that meet constraints efficiently and effectively. Future research could focus on further improving the algorithm and exploring its applicability in different educational contexts.

References:

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]
```