**CSE422**

**Artificial Intelligence**

**Lab**

8- Queen

* 8 X 8 2d board.
* Queens can be allocated that no queen can be attacked by another queen horizontally, vertically or diagonally

Task

1. Write a fitness function which checks the fitness of a board by checking the number of non-attacking pair of queens.   Hint: Maximum number of non-attacking pairs of queens can be (8\*7)/2. Input: board, Output: a number telling the fitness of the board
2. Write a Crossover function. A crossover function will take two boards, an index number as input and return two new boards.
3. Write a Mutation function.
4. Create a population of randomly generated boards.
5. Randomly select two members of the population.
6. Randomly generate an index number.
7. Call crossover function using the above two as input.
8. Call fitness function for the new boards from the output.
9. Call mutation function if necessary.
10. Add new members to a new population set if appropriate.
11. Run 5 to 10 until the new population set is large enough.
12. Select a few members from the old population to add to the new population set.
13. Run 1- 12 until a board with highest fitness value is created.

**Department of Computer Science and Engineering**

|  |  |
| --- | --- |
| **Course Code:CSE422** |  |
| **Course Name: Artificial Intelligence** | **Prerequisite:**CSE111, CSE221 |

**Lab 03**

**Genetic Algorithm**

1. **Lab Overview:**

The students will solve N-Queen problem using python programming and visualizing the evolution performance.

1. **Learning Objective:**
2. Introducing the 4-Queen problem
3. Solution of 4-Queen problem in Backtracking approach
4. Demerits of Backtracking approach
5. Introducing 8-Queen problem
6. Discussion on Genetic Algorithm
7. Solution of 8-Queen problem using GA
8. **Lesson Fit:**

There is pre-requisite to this lab: CSE111, CSE221. You should have intensive Programming Knowledge and capability to understand algorithms.

1. **Acceptance and Evaluation**

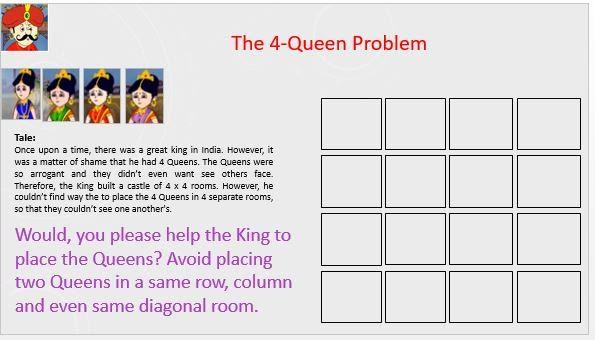
Students will show the output using different datasets and python code. They will be marked according to their lab performance. The main evaluation criteria will be based on project report and demonstration.

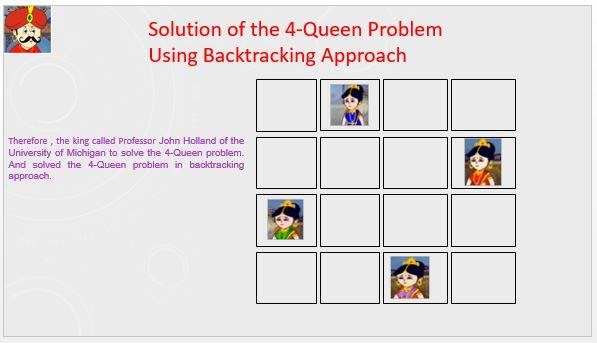
1. **Learning Outcome:**

After this lab, the students will be able to:

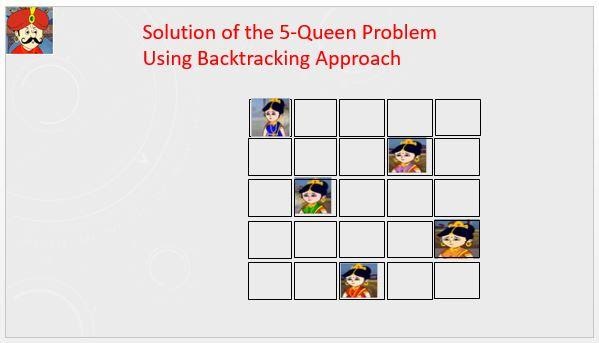
1. Demerits to solve N-Queen problem using Backtracking approach.
2. Solve the N-Queen problem using Genetic Algorithm
3. **Activity Detail**

* **Hour: 1.0 - 2.0**

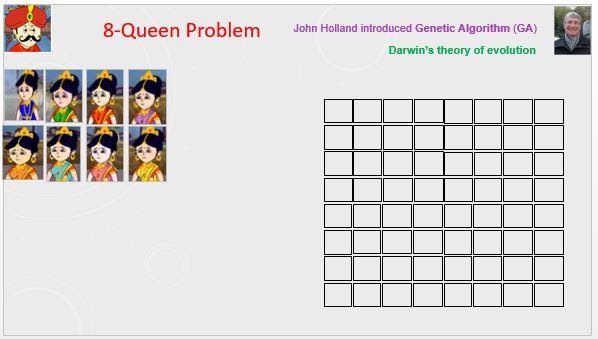


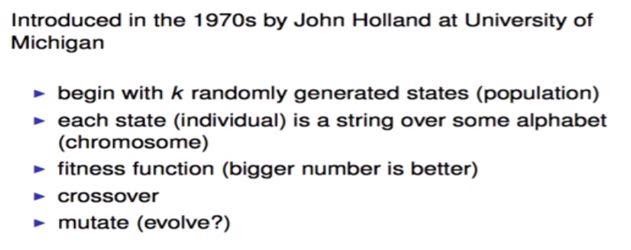


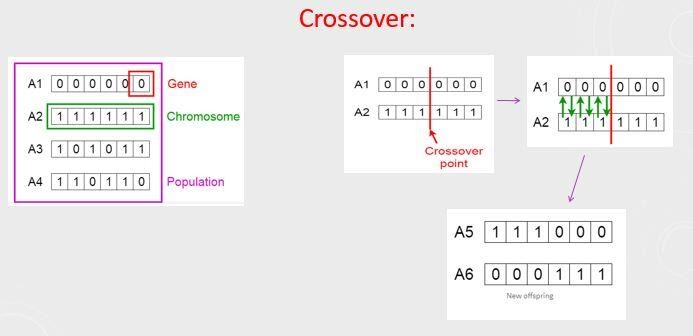


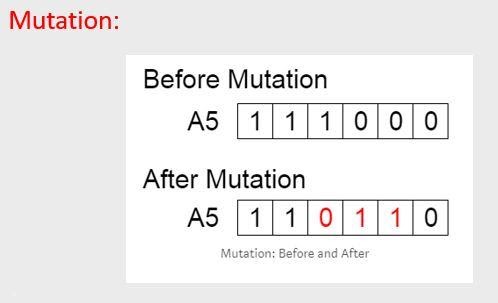


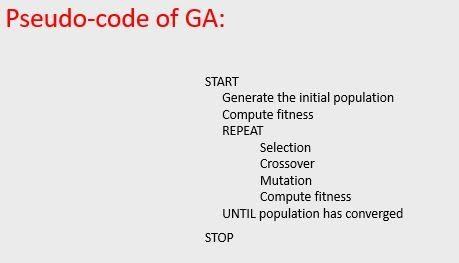


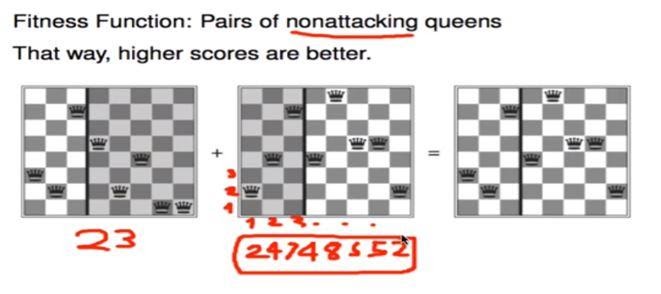


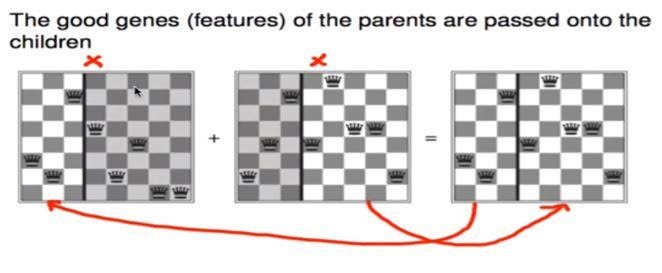


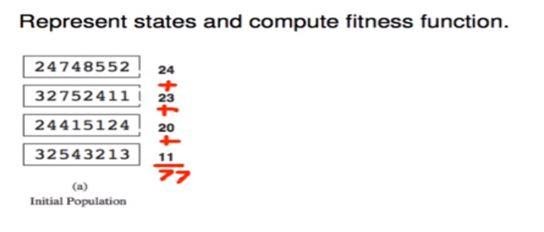


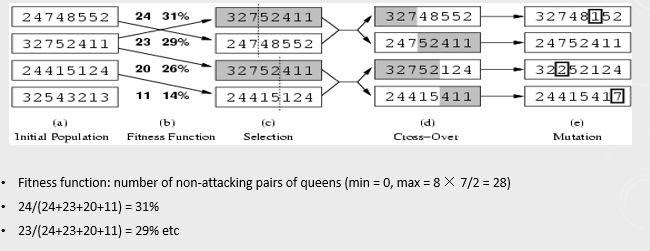


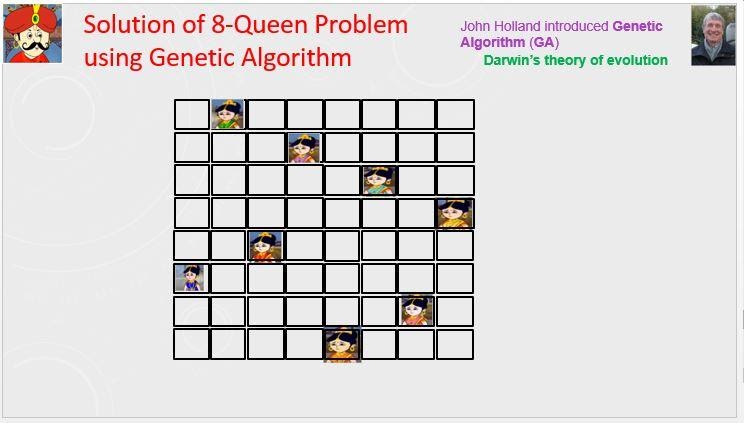


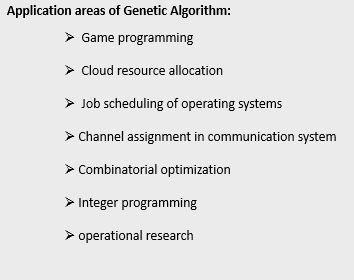












* **Hour: 2.0-3.0**

(It is Not a Group Task, Try Individually)

**Marks: 10**                                                                   **Time: 50 minutes**

**Task 1**: Implement N-Queen problem using Genetic Algorithm in python programming.

**Task 2**: Visualize the evolution through plotting the changes of fitness values, and the variances of fitness values for convergence.

Hints:  Take help from Prateek Joshi’s Book chapter 8, you can follow Covariance Matrix Adaptation Evolution Strategy (CMA-ES).

Evaluation Process (VIVA and Written answers): You have to explain your program and show your work to the Lab Instructor. Instructor may ask you some questions to evaluate your knowledge and expertise level.

---End---

Old Code:

Task01:

|  |
| --- |
| import random |
|  |  |
|  | def random\_chromosome(size): #making random chromosomes |
|  | return [ random.randint(1, nq) for \_ in range(nq) ] |
|  |  |
|  | def fitness(chromosome): |
|  | horizontal\_collisions = sum([chromosome.count(queen)-1 for queen in chromosome])/2 |
|  | diagonal\_collisions = 0 |
|  |  |
|  | n = len(chromosome) |
|  | left\_diagonal = [0] \* 2\*n |
|  | right\_diagonal = [0] \* 2\*n |
|  | for i in range(n): |
|  | left\_diagonal[i + chromosome[i] - 1] += 1 |
|  | right\_diagonal[len(chromosome) - i + chromosome[i] - 2] += 1 |
|  |  |
|  | diagonal\_collisions = 0 |
|  | for i in range(2\*n-1): |
|  | counter = 0 |
|  | if left\_diagonal[i] > 1: |
|  | counter += left\_diagonal[i]-1 |
|  | if right\_diagonal[i] > 1: |
|  | counter += right\_diagonal[i]-1 |
|  | diagonal\_collisions += counter / (n-abs(i-n+1)) |
|  |  |
|  | return int(maxFitness - (horizontal\_collisions + diagonal\_collisions)) #28-(2+3)=23 |
|  |  |
|  | def probability(chromosome, fitness): |
|  | return fitness(chromosome) / maxFitness |
|  |  |
|  | def random\_pick(population, probabilities): |
|  | populationWithProbabilty = zip(population, probabilities) |
|  | total = sum(w for c, w in populationWithProbabilty) |
|  | r = random.uniform(0, total) |
|  | upto = 0 |
|  | for c, w in zip(population, probabilities): |
|  | if upto + w >= r: |
|  | return c |
|  | upto += w |
|  | assert False, "Shouldn't get here" |
|  |  |
|  | def reproduce(x, y): #doing cross\_over between two chromosomes |
|  | n = len(x) |
|  | c = random.randint(0, n - 1) |
|  | return x[0:c] + y[c:n] |
|  |  |
|  | def mutate(x): #randomly changing the value of a random index of a chromosome |
|  | n = len(x) |
|  | c = random.randint(0, n - 1) |
|  | m = random.randint(1, n) |
|  | x[c] = m |
|  | return x |
|  |  |
|  | def genetic\_queen(population, fitness): |
|  | mutation\_probability = 0.03 |
|  | new\_population = [] |
|  | probabilities = [probability(n, fitness) for n in population] |
|  | for i in range(len(population)): |
|  | x = random\_pick(population, probabilities) #best chromosome 1 |
|  | y = random\_pick(population, probabilities) #best chromosome 2 |
|  | child = reproduce(x, y) #creating two new chromosomes from the best 2 chromosomes |
|  | if random.random() < mutation\_probability: |
|  | child = mutate(child) |
|  | print\_chromosome(child) |
|  | new\_population.append(child) |
|  | if fitness(child) == maxFitness: break |
|  | return new\_population |
|  |  |
|  | def print\_chromosome(chrom): |
|  | print("Chromosome = {}, Fitness = {}" |
|  | .format(str(chrom), fitness(chrom))) |
|  |  |
|  | if \_\_name\_\_ == "\_\_main\_\_": |
|  | nq = int(input("Enter Number of Queens: ")) #say N = 8 |
|  | maxFitness = (nq\*(nq-1))/2 # 8\*7/2 = 28 |
|  | population = [random\_chromosome(nq) for \_ in range(100)] |
|  |  |
|  | generation = 1 |
|  |  |
|  | while not maxFitness in [fitness(chrom) for chrom in population]: |
|  | print("=== Generation {} ===".format(generation)) |
|  | population = genetic\_queen(population, fitness) |
|  | print("") |
|  | print("Maximum Fitness = {}".format(max([fitness(n) for n in population]))) |
|  | generation += 1 |
|  | chrom\_out = [] |
|  | print("Solved in Generation {}!".format(generation-1)) |
|  | for chrom in population: |
|  | if fitness(chrom) == maxFitness: |
|  | print(""); |
|  | print("One of the solutions: ") |
|  | chrom\_out = chrom |
|  | print\_chromosome(chrom) |
|  |  |
|  | board = [] |
|  |  |
|  | for x in range(nq): |
|  | board.append(["x"] \* nq) |
|  |  |
|  | for i in range(nq): |
|  | board[nq-chrom\_out[i]][i]="Q" |
|  |  |
|  |  |
|  | def print\_board(board): |
|  | for row in board: |
|  | print (" ".join(row)) |
|  |  |
|  | print() |
|  | print\_board(board)  Task02:   |  | | --- | | import numpy as np | |  | import matplotlib.pyplot as plt | |  | from deap import algorithms, base, benchmarks, cma, creator, tools | |  |  | |  | # Function to create a toolbox | |  | def create\_toolbox(strategy): | |  | creator.create("FitnessMin", base.Fitness, weights=(-1.0,)) | |  | creator.create("Individual", list, fitness=creator.FitnessMin) | |  | toolbox = base.Toolbox() | |  | toolbox.register("evaluate", benchmarks.rastrigin) | |  | # Seed the random number generator | |  | np.random.seed(7) | |  | toolbox.register("generate", strategy.generate, creator.Individual) | |  | toolbox.register("update", strategy.update) | |  | return toolbox | |  |  | |  | if \_\_name\_\_ == "\_\_main\_\_": | |  | # Problem size | |  | num\_individuals = 10 | |  | num\_generations = 125 | |  | # Create a strategy using CMA-ES algorithm | |  | strategy = cma.Strategy(centroid=[5.0]\*num\_individuals, sigma=5.0, lambda\_=20\*num\_individuals) | |  | # Create toolbox based on the above strategy | |  | toolbox = create\_toolbox(strategy) | |  | # Create hall of fame object | |  | hall\_of\_fame = tools.HallOfFame(1) | |  |  | |  | # Register the relevant stats | |  | stats = tools.Statistics(lambda x: x.fitness.values) | |  | stats.register("avg", np.mean) | |  | stats.register("std", np.std) | |  | stats.register("min", np.min) | |  | stats.register("max", np.max) | |  |  | |  | logbook = tools.Logbook() | |  | logbook.header = "gen", "evals", "std", "min", "avg", "max" | |  |  | |  | # Objects that will compile the data | |  | sigma = np.ndarray((num\_generations, 1)) | |  | axis\_ratio = np.ndarray((num\_generations, 1)) | |  | diagD = np.ndarray((num\_generations, num\_individuals)) | |  | fbest = np.ndarray((num\_generations,1)) | |  | best = np.ndarray((num\_generations, num\_individuals)) | |  | std = np.ndarray((num\_generations, num\_individuals)) | |  |  | |  | for gen in range(num\_generations): | |  | # Generate a new population | |  | population = toolbox.generate() | |  | # Evaluate the individuals | |  | fitnesses = toolbox.map(toolbox.evaluate, population) | |  | for ind, fit in zip(population, fitnesses): | |  | ind.fitness.values = fit | |  | # Update the strategy with the evaluated individuals | |  | toolbox.update(population) | |  | # Update the hall of fame and the statistics with the | |  | # currently evaluated population | |  | hall\_of\_fame.update(population) | |  | record = stats.compile(population) | |  | logbook.record(evals=len(population), gen=gen, \*\*record) | |  | print(logbook.stream) | |  |  | |  | # Save more data along the evolution for plotting | |  | sigma[gen] = strategy.sigma | |  | axis\_ratio[gen] = max(strategy.diagD)\*\*2/min(strategy.diagD)\*\*2 | |  | diagD[gen, :num\_individuals] = strategy.diagD\*\*2 | |  | fbest[gen] = hall\_of\_fame[0].fitness.values | |  | best[gen, :num\_individuals] = hall\_of\_fame[0] | |  | std[gen, :num\_individuals] = np.std(population, axis=0) | |  |  | |  |  | |  | # The x-axis will be the number of evaluations | |  | x = list(range(0, strategy.lambda\_ \* num\_generations, strategy.lambda\_)) | |  | avg, max\_, min\_ = logbook.select("avg", "max", "min") | |  | plt.figure() | |  | plt.semilogy(x, avg, "--b") | |  | plt.semilogy(x, max\_, "--b") | |  | plt.semilogy(x, min\_, "-b") | |  | plt.semilogy(x, fbest, "-c") | |  | plt.semilogy(x, sigma, "-g") | |  | plt.semilogy(x, axis\_ratio, "-r") | |  | plt.grid(True) | |  | plt.title("blue: f-values, green: sigma, red: axis ratio") | |  |  | |  | plt.figure() | |  | plt.plot(x, best) | |  | plt.grid(True) | |  | plt.title("Object Variables") | |  | plt.figure() | |  | plt.semilogy(x, diagD) | |  | plt.grid(True) | |  | plt.title("Scaling (All Main Axes)") | |  | plt.figure() | |  | plt.semilogy(x, std) | |  | plt.grid(True) | |  | plt.title("Standard Deviations in All Coordinates") | |  | plt.show() | |  |  | |  |  | |
|  |  |