Lab - 4

Question 1

The 8-queens problem is about placing 8 queen chess pieces on an 8x8 chessboard in such a way that none of the pieces are under attack. The goal is to produce a board state where any pair cannot lie on the same row, column, or diagonal.

Tasks – Prepare a report

Explain how this problem can be solved using an evolutionary algorithm by discussing

- the genetic operators: encoding of phenotype and genotype.
- the fitness function.
- the mutation, the crossover.
- the selection

Genetic Operators

The phenotype represents the board configuration itself. The structure of the board including its shape and the number of tiles will never change, as well as the rules of what spaces the queen can occupy on its turn. The 8 rows and 8 columns will be represented as a 2-dimensional array with values 0 to 7.

The genotype represents the positions of the 8 queens. These will be represented as a linear 1-dimensional array that's a string of numbers. Each index of the array represents the column number, and the value at each index represents the row number. So, if all pieces were placed on the top row, the string value would be 00000000.

Figure 1: Genetic operator examples

The population will be initialised with 128 boards with randomly assigned values.

Fitness Function

The fitness function will be evaluating the state of the board and counting the number of queens that aren't attacking another queen. I will be using a reward and penalty method to set and evaluate the fitness score.

The fitness score will start at 0. Each pair of non-attacking queens will increase the fitness score by 1. The problem will be solved once the fitness score reaches 28 (as the number of pairs would be 8 * 7 / 2), meaning that all 8 queens aren't attacking another queen.

Mutation and Crossover

I will be using recombination to produce new offspring which will again be part of the process. The crossover will be 1-point crossover, with the crossover point being after the third value.

The first child will have the first 3 values from parent 1 and the last 5 values from parent 2, the second child will have the last 5 values from parent 2.

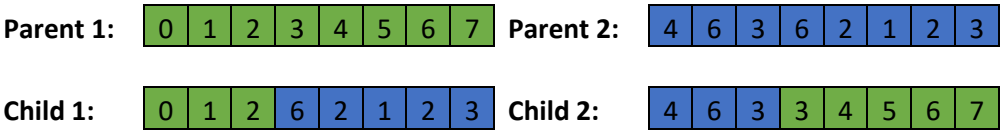

Figure 2: Crossover example

Each child has an 80% chance of being mutated. If they are selected for mutation, a random value in the array will change to a random value between 0 and 7.

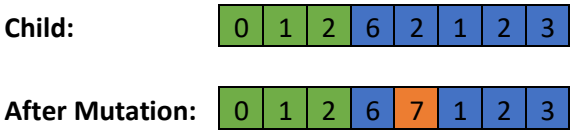

Figure 3: Mutation example

The Selection

There will be two parents producing two children while keeping the population size at 128. This means that each iteration will be selecting 64 survivors out of the population size. The 64 survivors will be randomly selected, but the individuals with a higher fitness score have an increased chance of being selected.

This chance is determined by roulette wheel selection. The total fitness score is calculated and each individual and its fitness score is put in a random order. A random number between 1 and the total fitness score will be generated and the first individual whose fitness score plus the total score of the individuals preceding it is greater than or equal to the random number is selected as a survivor. This is repeated until there are 64 survivors who then become parents.

These 64 parents are put in pairs that will each produce 2 offspring, bringing the total population back up to 128.

This whole process is repeated until an individual with a fitness score of 28 is found, and that individual will return the board state with the solution.

Question 2

Explain your understanding about how Evolutionary Algorithm has been implemented and discuss its working.

The evolutionary strategies algorithm is inspired by evolution and natural selection. There's no crossover in this algorithm, only mutation.

A population of candidate solutions are randomly generated. These solutions are evaluated and the best are kept on and used as a basis for generating new candidate solutions via mutation. These new solutions can either fully replace the parents in its next generation, or be appended to a subset of the parents and evaluated again in the next generation.

The size of the population is referred to as *lambda* and the number of parents is referred to as *mu*. The number of children that are created is calculated by dividing lambda with mu.

(mu, lambda) – The comma indicates that the children replace the parents.

(mu + lambda) – The plus indicates that the children are appended to a subset of the parents.

The Ackley function is used as a global optimisation technique, with its objective function being [0,0].

```
def objective(v):
    x, y = v
    return -20.0 * exp(-0.2 * sqrt(0.5 * (x**2 + y**2))) - exp(0.5 * (cos(2
* pi * x) + cos(2 * pi * y))) + e + 20
```

The in_bounds() function returns True if the solution (called point in the parameters) falls within the search space (called bounds in parameters). Returns False if it's not within the search space. This function is used when generating a new population.

```
bounds = asarray([[-5.0, 5.0], [-5.0, 5.0]])
# check if a point is within the bounds of the search
def in_bounds(point, bounds):
    # enumerate all dimensions of the point
    for d in range(len(bounds)):
        # check if out of bounds for this dimension
        if point[d] < bounds[d, 0] or point[d] > bounds[d, 1]:
            return False
    return True
```

The best and best_eval variables get initialised.

```
# evolution strategy (mu, lambda) algorithm
def es_comma(objective, bounds, n_iter, step_size, mu, lam):
    best, best_eval = None, 1e+10
```

The number of children are calculated by dividing lambda by mu. This is stored in n_children.

```
# number of parents selected
mu = 1
# the number of children generated by parents
lam = 4
# calculate the number of children per parent
```

```
    n_children = int(lam / mu)
```

Each value in the range of the population gets assigned a random value that's within the range of the bounds array and then gets appended to a list called population.

```
    # initial population
    population = list()
    for _ in range(lam):
        candidate = None
        while candidate is None or not in_bounds(candidate, bounds):
            candidate = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] -
bounds[:, 0])
        population.append(candidate)
```

For each epoch in the range of the number of iterations, the candidate solutions get passed through the objective function to calculate its score and gets stored in a list called scores. This list is then put in ascending order (as this is a minimisation problem with the global optimum of Ackley's function being [0,0]) and stored in a variable called ranks.

```
# define the total iterations
n_iter = 5000
    # perform the search
    for epoch in range(n_iter):
        # evaluate fitness for the population
        scores = [objective(c) for c in population]
        # rank scores in ascending order
        ranks = argsort(argsort(scores))
```

The candidate solutions stored in ranks are then selected to be parents based on their rank. The number selected is dependent on the mu variable. At the moment it's set to 1, so only the best parent gets selected.

```
        # select the indexes for the top mu ranked solutions
        selected = [i for i,_ in enumerate(ranks) if ranks[i] < mu]
```

The scores of the parents are checked to see if it's the best solution seen so far and prints the result.

```
# create children from parents
        children = list()
        for i in selected:
            # check if this parent is the best solution ever seen
            if scores[i] < best_eval:
                best, best_eval = population[i], scores[i]
                print('%d, Best: f(%s) = %.5f' % (epoch, best, best_eval))
```

The children are created as modified versions of the parents. How different the random value is depends on the step_size variable. The lower the step size, the smaller the difference.

```
# define the maximum step size
step_size = 0.15
            # create children for parent
            for _ in range(n_children):
                child = None
                while child is None or not in_bounds(child, bounds):
```

```
                    child = population[i] + randn(len(bounds)) * step_size
                children.append(child)
```

The children then become the population and the best solution and score is returned.

```
        population = children
    return [best, best_eval]
```

In the es_plus function, the only difference is that the children are appended to the parents and set as the new population instead of replacing them. Both the parents and children are a part of the new population in each iteration.

```
            # keep the parent
            children.append(population[i])
```

This is a more greedy behaviour that poses a risk of getting stuck to the local optima resulting in a suboptimal solution. However, focusing on only the good candidate solutions can help it find even more precise optimal solutions within a good region.