assignment_2

April 22, 2024

1 Assignment 2

Assignment 2: Evolutionary Algorithms

Goal: Implement an Evolutionary Algorithm to solve continuous and discrete problems.

- Part1: continuous problem that is concerned with finding minimum of functions, in this case, Sphere and Ackley functions.
- Part 2: discrete problem that is concerned with finding the solution for the N-queens problem. Could be defined as minimization or maximization.

For both parts, this assignment requires implementation of the main components of an evolutionary algorithm (i.e. *crossover*, *mutation*, *parent selection*, etc), and construction of your algorithm to solve given problems.

Please answer the `Questions` and implement coding `Tasks` by filling **PLEASE FILL IN** sections. *Documentation* of your code is also important. You can find the grading scheme in implementation cells.

- Plagiarism is automatically checked and set to **0 points**
- It is allowed to learn from external resources but copying is not allowed. If you use any external resource, please cite them in the comments (e.g. `# source: https://...../` (see `fitness_function`))

Setup

Install Prerequisites (Part 1 and 2)

[31]:
```
# Run this cell to install the required libraries
%pip install numpy matplotlib scipy
```

```
Requirement already satisfied: numpy in
/Users/selma/anaconda3/envs/da/lib/python3.11/site-packages (1.25.2)
Requirement already satisfied: matplotlib in
/Users/selma/anaconda3/envs/da/lib/python3.11/site-packages (3.8.2)
Requirement already satisfied: scipy in
/Users/selma/anaconda3/envs/da/lib/python3.11/site-packages (1.12.0)
Requirement already satisfied: contourpy>=1.0.1 in
/Users/selma/anaconda3/envs/da/lib/python3.11/site-packages (from matplotlib)
(1.0.5)
Requirement already satisfied: cycler>=0.10 in
```

```
/Users/selma/anaconda3/envs/da/lib/python3.11/site-packages (from matplotlib)
(0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in
/Users/selma/anaconda3/envs/da/lib/python3.11/site-packages (from matplotlib)
(4.25.0)
Requirement already satisfied: kiwisolver>=1.3.1 in
/Users/selma/anaconda3/envs/da/lib/python3.11/site-packages (from matplotlib)
(1.4.4)
Requirement already satisfied: packaging>=20.0 in
/Users/selma/anaconda3/envs/da/lib/python3.11/site-packages (from matplotlib)
(23.0)
Requirement already satisfied: pillow>=8 in
/Users/selma/anaconda3/envs/da/lib/python3.11/site-packages (from matplotlib)
(9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/Users/selma/anaconda3/envs/da/lib/python3.11/site-packages (from matplotlib)
(3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in
/Users/selma/anaconda3/envs/da/lib/python3.11/site-packages (from matplotlib)
(2.8.2)
Requirement already satisfied: six>=1.5 in
/Users/selma/anaconda3/envs/da/lib/python3.11/site-packages (from python-
dateutil>=2.7->matplotlib) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

Imports (Part 1 and 2)

[32]:
```
# Necessary libraries
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import ranksums

# Set seed
np.random.seed(42)
```

Jupyter Notebook Magic (Part 1 and 2)

[33]:
```
# Enables inline matplotib graphs
#%matplotlib inline

# Comment the line above and uncomment the lines below to have interactive plots
# WARN: may cause dependency issues

plt.ion()
plt.show()
```

1.1 Part 1: Continuous Optimization (5 points total)

In this part of the assignment you will implement an Evolutionary Algorithm to find the minimum of the following functions: Sphere and Ackley functions.

Function Definitions & Plotting

Sphere Function

[34]:
```
def Sphere(x):
    """source: https://www.sfu.ca/~ssurjano/spheref.html"""
    dimension = x.shape[0]
    return (1 / dimension) * (sum(x**2))
```

Ackley Function

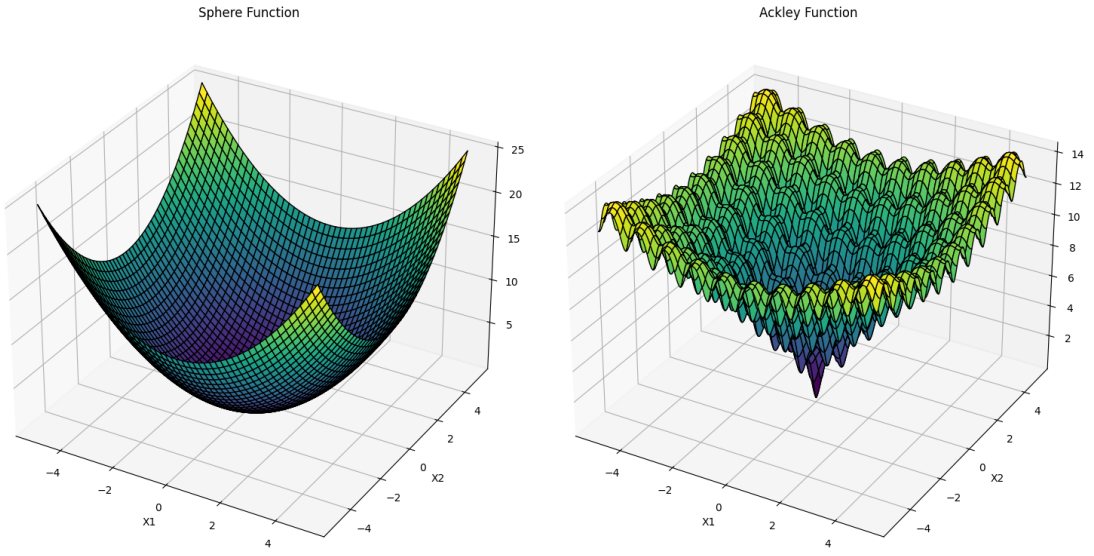
[35]:
```
def Ackley(x):
    """source: https://www.sfu.ca/~ssurjano/ackley.html"""

    # Ackley function parameters
    a = 20
    b = 0.2
    c = 2 * np.pi
    dimension = len(x)

    # Individual terms
    term1 = -a * np.exp(-b * np.sqrt(sum(x**2) / dimension))
    term2 = -np.exp(sum(np.cos(c * xi) for xi in x) / dimension)
    return term1 + term2 + a + np.exp(1)
```

Plotting

[36]:
```
# Generate data for plotting
boundary_point, resolution = 5, 500
x = np.linspace(-boundary_point, boundary_point, resolution)
y = np.linspace(-boundary_point, boundary_point, resolution)

# Generate the coordinate points
X, Y = np.meshgrid(x, y)
positions = np.column_stack([X.ravel(), Y.ravel()])

# Get depths for all coordinate positions
z_unimodal = np.array(list(map(Sphere, positions))).reshape([resolution,␣
 ↪resolution])
z_multimodal = np.array(list(map(Ackley, positions))).reshape([resolution,␣
 ↪resolution])
```

[37]:
```
# Create 3D plot
fig = plt.figure(figsize=(15, 8))
```

```
titles = ["Sphere Function", "Ackley Function"]
for idx, z in enumerate([z_unimodal, z_multimodal]):
    # Create sub-plot
    ax = fig.add_subplot(121 + idx, projection="3d")

    # Plot the surface
    ax.plot_surface(X, Y, z, cmap="viridis", edgecolor="k")

    # Set labels
    ax.set_xlabel("X1")
    ax.set_ylabel("X2")
    ax.set_title(titles[idx])
    # ax.autoscale(True)

# Show the plot
plt.tight_layout()
plt.show()
```



Question 1.1 (0-0.2 pt): Looking at the Sphere and Ackley functions, please discuss the characteristics of these functions and compare their complexity in terms of local and global optima.

Answer:

The Sphere function has d local minima, one along each dimension. It has one global minimum and its continuous, convex and unimodal. The fucntion is more simple than Ackley. Landscpae is relatively simple and smooth.

Ackley Function on the other hand more comlex. It has a nearly flat outer region and a large hole at the centre. There are multiple peaks and valleys in the landscape of the function. Since it has numerious local minima, its quite challenging for optimizers, since there is great chance that optimizer could be stuck at some logac minima for a minimalizaiton problem.

Finding global optima in Sphere function would be easier comparing to Ackley Function, since Ackley has super complex landscape and optimizer could get easily stuck at some local minima before finding the optima.

Question 1.2 (0-0.25-0.5 pt): Please provide a pseudo-code for your evolutionary algorithm. Please try to be as formal as possible!

Answer:

```
Function initialization(population_size, num_dimensions):
    Initialize x as empty list
    For each individual in population_size:
        Randomly generate an individual values between -50 and 50 with size num_dimensions
        Append individuals to list x
    Return x

Function evaluation(x, objective_function):
    Initialize fitness as an empty list
    For each individual in x:
        Call objective_function to calculate objective_value
        Set objective_value as fitness_score
        Append fitness_score to list fitness
    Return fitness

Function crossover(x_parents, p_crossover):
    Randomly shuffle x_parents
    Initialize offspring as empty list
    For i from 0 to length of x_parents - 1 by 2:
        If random number < p_crossover:
            Generate a crossover mask for genes
            Create offspring1 by combining x_parents[i] and x_parents[i+1] based on the mas
            Create offspring2 by combining x_parents[i+1] and x_parents[i] based on the mas
            Append offspring1 and offspring2 to offspring
        Else:
            Append x_parents[i] and x_parents[i+1] to offspring
    Return offspring

Function mutation(population, mutation_rate):
    For each gene in individual from population:
            If random number < mutation_rate:
                Set perturbation to normally distributed random value with mean 0 and stand
                gene += perturbation
    Return population
```

```
Function parent_selection(x, f):
    Initialize x_parents  as empty lists
    Initialize f_parents as empty lists
    Set population_size equal to length of x
    Set tournament_size to 2
    For each individual in the population:
        Select tournament_size random individuals from x
        Determine winner with the lowest(minimization problem) fitness
        Append the winner to x_parents
        Append the winner to f_parents
    Return x_parents, f_parents

Function survivor_selection(x, f, x_offspring, f_offspring):
    Combine x and x_offspring into combined_population
    Combine f and f_offspring into combined_fitness
    Set population_size to the length of x
    Sort combined_population based on combined_fitness in ascending order
    Clear x
    Clear f
    For each index in the first 'population_size' elements of 'sorted_indices':
        Append the element at this index from 'combined_population' to 'x'
        Append the element at this index from 'combined_fitness' to 'f'
    Return x, f

Function ea(population_size, max_fit_evals, p_crossover, m_rate, dimensions, objective_fun
    ################################################################
    Assign x by calling initialization function
    Assign f by calling evaluation function
    ################################################################

    ################################################################
    Selecting x_parents and f_parents by calling parent_seelction function
    Generate x_offspring by calling crossover function
    Evaluate f_offspring by calling evaluation function

    Set x_mutated by iterating through individuals in x_offspring for mutation function
    Evaluate f-mutated by calling evaluation function

    Select x and f by calling survivor_selection function
    ################################################################
```

Note: for function ea, only the parts with 'PLEASE FILL IN' considered in pseudocode.

Task 1.1: Implementation of Evolutionary Algrotihm (0-0.65-1.3-1.95-2.6 pt): Implement an evolutionary algorithm and its components to find the minimum point of a function. Here, domain should be between [-50,50].

[38]:
```
##################################
# Grading
# 0 pts if the code does not work, code works but it is fundamentally incorrect
# 0.65 pts if the code works but some functions are incorrect and it is badly␣
 ↪explained
# 1.3 pts if the code works but some functions are incorrect but it is␣
 ↪explained well
# 1.95 pts if the code works very well aligned with the task without any␣
 ↪mistakes, but it is badly explained
# 2.6 pts if the code works very well aligned with the task without any␣
 ↪mistakes, and it is well explained
#####################################################################


#Initialize a population randomly based on the population size and dimensions
def initialization(population_size, num_dimensions):
    """
    Initialize the starting population with random individuals.
    Each gene of an individual corresponds to a point on a dimension in the␣
 ↪function
    """

    ###################################################################

    x = []

    # Iterate through the population_size to give each individual a point on␣
 ↪the dimention in the function
    for _ in range(population_size):

        #Draw samples from a uniform distribution for each individual.
        individual = np.random.uniform(low=-50, high=50, size=num_dimensions)

        x.append(individual) #Appending all the points in x.

    ###################################################################

    return x #return population


# Implement the evaluation function that can evaluate all the solutions in a␣
 ↪given population.
def evaluation(x, objective_function):
    """Evaluate the fitness of the population members"""

    ###################################################################
```

```
    """This function aims to find the fitness score of the individuals with␣
 ↪using objective functions"""

    fitness = []

    # Iterating over all the individuals in x
    for individual in x:
        # Calculating onjective value of each individual
        objective_value = objective_function(individual)

        # Since this is a minimization problem, lower objective value is better.
 ↪
        fitness_score = objective_value

        # Appending fitness scores
        fitness.append(fitness_score)

    ######################################################################
    return fitness


# Implement the crossover operator by choosing a suitable method. For␣
 ↪inspiration, take a look at the lecture slides
def crossover(x_parents, p_crossover):
    """Perform crossover to create offsprings."""

    ######################################################################
    np.random.shuffle(x_parents)  # Shuffle parents to ensure random pairing
    offspring = []

    """Performing Uniform Crossover. Because considering the landscape of␣
 ↪Sphere and Ackley, and
    increased dimensions there should be more explorations has to be made. This␣
 ↪is provided with crossover and generated
    variations."""

    for i in range(0, len(x_parents) - 1, 2):

        #using crossover probability to make offsprings.
        #p_crossover=0.9 so there is 90% chance that a crossover will occur
        if np.random.rand() < p_crossover:
            # Proceed with uniform crossover
            # Uniformly decide genes to swap, a 50% chance for each gene
            mask = np.random.rand(len(x_parents[i])) > 0.5 #mask is a boolean␣
 ↪array, it gives for each index either true or false.
```

```
            offspring1 = np.where(mask, x_parents[i], x_parents[i+1]) #if mask␣
 ↪is True, the gene is taken from  x_parents[i]; else from x_parents[i+1]
            offspring2 = np.where(mask, x_parents[i+1], x_parents[i]) #if mask␣
 ↪is True, the gene is taken from  x_parents[i+1]; else from x_parents[i]
            offspring.extend([offspring1, offspring2])
            #print(offspring1, offspring2, offspring )
        else:
            # No crossover, just pass parents to the next generation
            offspring.extend([x_parents[i], x_parents[i+1]])

    ######################################################################

    return offspring



# Implement the crossover operator by choosing a suitable method. For␣
 ↪inspiration, take a look at the lecture slides
def mutation(x, mutation_rate):
    """Apply mutation to an individual."""

    ######################################################################

    # Iterate through each gene in the individual
    for i in range(len(x)):
        # Checking if the mutation should occur or not
        if np.random.rand() < mutation_rate:
            # adding a small random perturbation to make the mutation
            perturbation = np.random.normal(loc=0, scale=1) #normally␣
 ↪distributed
            x[i] += perturbation

    ######################################################################

    return x


def parent_selection(x, f):
    """Select parents for the next generation"""

    ######################################################################

    x_parents = []
    f_parents = []
    population_size = len(x)
    tournament_size = 2 # The number of individuals to compete in each␣
 ↪tournament
```

```
    """Tournament Selection method."""
    for _ in range(population_size):
        # Randomly select tournament_size individuals from the population
        indices = np.random.choice(population_size, tournament_size,␣
↪replace=False) #Gives two indices [a, b] for tournament
        #x[i] = [genes], f[i] = fitness score of that array
        tournament_individuals = [(x[i], f[i]) for i in indices] #Contains two␣
↪arrays on the given indices(each time randomly)

        # Select the best individual from the tournament
        #the best fitness value for a population is the smallest fitness value␣
↪for any individual in the population.
        winner = min(tournament_individuals, key=lambda individual:␣
↪individual[1])

        # Appending the winner to the parent list
        x_parents.append(winner[0])
        f_parents.append(winner[1])

    ######################################

    return x_parents, f_parents


def survivor_selection(x, f, x_offspring, f_offspring):
    """Select the survivors, for the population of the next generation"""


    ######################################################################

    """Truncation selection method (form of elitism ), where only the best␣
↪individuals with the best fitness survive to the next generation."""

    # Put the current population and offsprings into one extended pool
    combined_population = x + x_offspring
    combined_fitness = f + f_offspring

    # Determine the number of individuals to retain (assuming population size␣
↪remains constant)
    population_size = len(x)

    # Get the indices of the individuals with the highest fitness scores
    # Sort by fitness ascendingly for minimization
    sorted_indices = sorted(range(len(combined_fitness)), key=lambda i:␣
↪combined_fitness[i])
```

```
    # Clear the original population lists
    x.clear()
    f.clear()
    # Fill the original lists with the top individuals based on sorted indices
    for index in sorted_indices[:population_size]:
        x.append(combined_population[index])
        f.append(combined_fitness[index])

    ###################################################################

    return x, f


def ea(
    # hyperparameters of the algorithm
    population_size,
    max_fit_evals,  # Maximum number of evaluations
    p_crossover,  # Probability of performing crossover operator
    m_rate,  # mutation rate
    dimensions,  # number of dimensions
    objective_function,  # objective function to be minimized
):
    # Calculate the maximum number of generations
    # Maximum number of function evaluations should be the same independent of␣
 ↪the population size
    max_generations = int(max_fit_evals / population_size)  # DO NOT CHANGE



    ###################################################################

    #Initializing the population and evaluating the fitness.
    x = initialization(population_size, dimensions)
    f = evaluation(x, objective_function)

    ###################################################################

    # Find the best individual and append to a list to keep track in each␣
 ↪generation
    idx = np.argmin(f)
    x_best = [x[idx]]
    f_best = [f[idx]]

    # Loop over the generations
    for _ in range(max_generations - 1):
        # Perform the EA steps
```

```
        ######################################################################
        """
        1. Selecting the parents based on fitness (tournement method)
        2. Generate offsprings with the crossover
        3. Evaluate offsprings after the crossover
        4. Take the x_offspring and mutate
        5. Re-evaluate after mutation
        6. Survivor selection for the next generations

        """
        # Selecting the parents
        x_parents, f_parents = parent_selection(x, f)

        # Generate offsprings with the crossover
        x_offspring = crossover(x_parents, p_crossover)
        f_offspring = evaluation(x_offspring, objective_function)  # Evaluate␣
↪offspring

        # Mutation of offspring
        x_mutated = [mutation(individual, m_rate) for individual in x_offspring]
        f_mutated = evaluation(x_mutated, objective_function)  # Re-evaluate␣
↪after mutation

        # Survivor selection to form the new generation
        x, f = survivor_selection(x, f, x_mutated, f_mutated)

        ######################################################################


        # Find the best individual in current generation and add to the list
        idx = np.argmin(f)
        xi_best = x[idx]
        fi_best = f[idx]
        if fi_best < f_best[-1]:
            x_best.append(xi_best)
            f_best.append(fi_best)
        else:
            x_best.append(x_best[-1])
            f_best.append(f_best[-1])

    return x_best, f_best  # return the best solution and fitness in each␣
↪generation
```

Check Your Implementation: Running The Evolutionary Algorithm Run the cell below, if you implemented everything correctly, you should see the algorithm running.

[39]:
```
# Dummy parameters
kwargs = {
    "population_size": 20,
    "max_fit_evals": 1000,  # maximum number of fitness evaluations
    "p_crossover": 0.9,  # crossover probability
    "m_rate": 0.1,  # mutation rate
    "dimensions": 10,
    "objective_function": Sphere,
}

# Run the EA
x_best, f_best = ea(**kwargs)

# Print the best individual and its fitness
print("Best solution:", x_best[-1])
print("Best Fitness:", f_best[-1])


# Clear cache
del x_best, f_best, kwargs
```

```
Best solution: [-0.76112811  9.20529693  8.60272307  0.26619662 -2.53783945
-8.82400126
 -9.76200615  2.32011776 -3.85989081 -2.01043994]
Best Fitness: 36.331847633239434
```

Reults and statistical analysis

Remember that the EAs are sthocastic algorithms that can produce different results as a result of independent runs.

How do we find overal performance of the algorithm and compare the results?

By running multiple times and performing statitical tests. Therefore, you would need to run your algorithm **20 times** and plot the *average* results.

First, we would need to **defining some helper functions** for finding the average and standard deviations of multiple runs and ploting them. In the next few cells, we give you some pre-made functions for this purpose.

There is no work for you to do, but do look over them and get familiar with how they operate.

[40]:
```
def calculate_mean_std(f_best):
    """This is a helper function to calculate the mean and standard deviation␣
 ↪of the best fitness values."""
    f_best = np.array(f_best)
    avg = np.mean(f_best, axis=0)
    std = np.std(f_best, axis=0)
    return avg, std
```

[41]:
```
def run_experiment(population_size, p_crossover, m_rate):
    runs = 20  # DO NOT CHANGE - number of runs
    max_fit_evals = 5000  # DO NOT CHANGE

    sphere10D = []
    sphere50D = []
    ackley10D = []
    ackley15D = []

    for _ in range(runs):
        _, f_best_sphere10D = ea(
            population_size[0],
            max_fit_evals,
            p_crossover[0],
            m_rate[0],
            10,
            Sphere,
        )
        _, f_best_sphere50D = ea(
            population_size[1],
            max_fit_evals,
            p_crossover[1],
            m_rate[1],
            50,
            Sphere,
        )
        _, f_best_ackley10D = ea(
            population_size[2],
            max_fit_evals,
            p_crossover[2],
            m_rate[2],
            10,
            Ackley,
        )
        _, f_best_ackley15D = ea(
            population_size[3],
            max_fit_evals,
            p_crossover[3],
            m_rate[3],
            15,
            Ackley,
        )

        sphere10D.append(f_best_sphere10D)
        sphere50D.append(f_best_sphere50D)
        ackley10D.append(f_best_ackley10D)
        ackley15D.append(f_best_ackley15D)
```

```
    # find average and std of the runs
    sphere10D_avg, sphere10D_std = calculate_mean_std(sphere10D)
    sphere50D_avg, sphere50D_std = calculate_mean_std(sphere50D)
    ackley10D_avg, ackley10D_std = calculate_mean_std(ackley10D)
    ackley15D_avg, ackley15D_std = calculate_mean_std(ackley15D)

    avgs = [sphere10D_avg, sphere50D_avg, ackley10D_avg, ackley15D_avg]
    stds = [sphere10D_std, sphere50D_std, ackley10D_std, ackley15D_std]
    all_runs = [
        sphere10D,
        sphere50D,
        ackley10D,
        ackley15D,
    ]

    return avgs, stds, all_runs
```

[42]:
```
def generate_subplot_function(
    avgs_experiment_1,
    stds_experiment_1,
    labels,
    avgs_experiment_2,
    stds_experiment_2,
    n_columns,
    n_queens,
):
    """This helper function generates subplots for the experiments."""
    fig, axes = plt.subplots(nrows=1, ncols=n_columns, figsize=(18, 6))

    for i in range(len(avgs_experiment_1)):
        if avgs_experiment_2 is not None:
            # Plot data for subplot 1
            axes[i].plot(avgs_experiment_2[i], label="Experiment 2",␣
 ↪color="green")
            axes[i].fill_between(
                np.arange(len(avgs_experiment_2[i])),
                avgs_experiment_2[i] - stds_experiment_2[i],
                avgs_experiment_2[i] + stds_experiment_2[i],
                alpha=0.2,
                color="green",
            )
            axes[i].set_ylim(bottom=0)

            if n_queens:
                axes[i].set_ylim(top=n_queens[i])
```

```
        axes[i].plot(avgs_experiment_1[i], label="Experiment 1", color="blue")
        axes[i].fill_between(
            np.arange(len(avgs_experiment_1[i])),
            avgs_experiment_1[i] - stds_experiment_1[i],
            avgs_experiment_1[i] + stds_experiment_1[i],
            alpha=0.2,
            color="blue",
        )
        axes[i].set_title(labels[i])
        axes[i].set_ylim(bottom=0)
        if n_queens:
            axes[i].set_ylim(top=n_queens[i])

    # Set common labels and title
    for ax in axes:
        ax.set_xlabel("Generations")
        ax.set_ylabel("Average Best Fitness")
        ax.legend()

    plt.tight_layout()
```

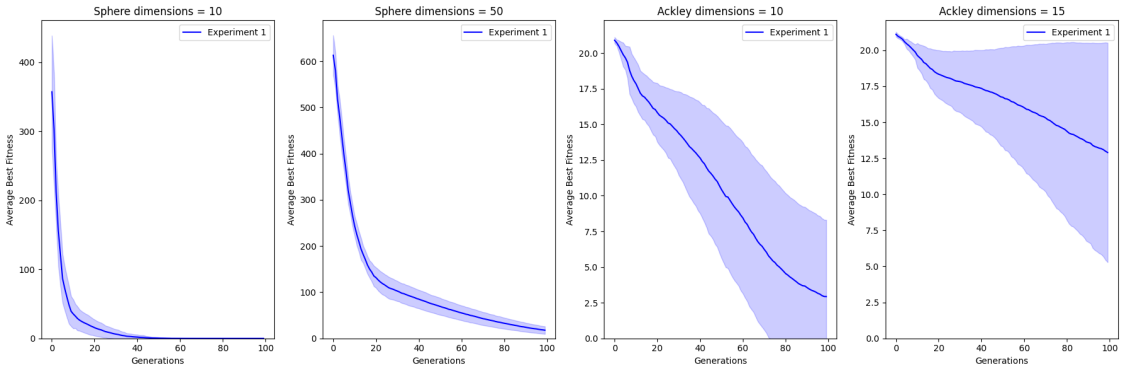
Running The Experiments In the following cell we run the EA over several different hyper-parameter values.

[43]:
```
population_size = [50, 50, 50, 50]  # DO NOT CHANGE
p_crossover = [0.8, 0.8, 0.8, 0.8]  # DO NOT CHANGE
m_rate = [0.1, 0.1, 0.1, 0.1]  # DO NOT CHANGE

avgs_experiment_1, stds_experiment_1, all_runs_experiment_1 = run_experiment(
    population_size, p_crossover, m_rate
)
```

Plotting The Results In the following cell we plot the results of the experiments.

[44]:
```
labels = [  # DO NOT CHANGE
    "Sphere dimensions = 10",
    "Sphere dimensions = 50",
    "Ackley dimensions = 10",
    "Ackley dimensions = 15",
]

generate_subplot_function(
    avgs_experiment_1,
    stds_experiment_1,
    labels,
    avgs_experiment_2=None,
```

```
    stds_experiment_2=None,
    n_columns=4,
    n_queens=None,
)
```



Question 1.3 (0-0.25-0.5 pt): Describe the results that you see in the line graphs. How is the performance of the EA on Sphere and Ackley functions? How the results are different between functions and dimensions? What causes these differences?

Answer:

The graphs in the figure represent how the average best fitness—the mean of the best fitness scores obtained from multiple runs of the EA, with lower values indicating more optimal solutions—of the genes improves over the generations and finds the optimum solution for the minimization problem. To illustrate how the graphs work; for 'Sphere dimension = 10', the average best fitness for the solution is around 0 by 40th generation, whereas for 'Sphere dimension = 50', by the 40th generation, it is around 90.

As the dimensions increase for the Sphere function, the average best fitness also changes over the generations. For instance, at the 100th generation, it is around 20 for dimension 50. With the higher dimensions, the complexity of the problem increases. Additionally, since the search space in 50 dimensions is expanded, the EA optimizer has to search a larger area, and thus there are many more solutions to be evaluated, and regions to be explored. In this case, the optimizer samples more points to gain the same level of understanding of the problems landscape, leading to higher fitness values because it has explored **less** of the search space effectively. The steepness of the line on the Sphere graphs indicates a quicker convergence, as can be seen from the dimension = 10 of Sphere which shows that the EA consistently finds better solutions more quickly for 10 d.

This is also observed with the Ackley function; it is more complex compared to the Sphere function and has multiple local minima but since the nature of the function is super complex it definitely challenges algorithm's performance. Furthermore, since the Ackley function has numerous local minima, within the different runs,1 the EA may get stuck in these minima. This results in a wide range of fitness values, as seen in the figure by the large light blue areas surrounding the blue lines in the 3rd and 4th graphs.

Despite the higher dimensionality of the Sphere function, which challanges the algorithm with the large search space, Sphere function's graph showcases a steep decline in fitness which indicates that the significant progress of the algorithm within the first few generations. On the other hand, the Ackley function, with fewer dimensions, shows the algorithm converging towards lower fitness values more rapidly.

Improve Your Results

Experiment with the hyperparameters of the algorithm and find a set of parameters that can perform better than the previous results. Compare the results using statistical test and find a settings where there is a statistically significant improvement.

You can adjust the *population size*, *crossover probability* and *mutation rate* to find the settings that can work statistically better relative to the previous results. Please look at the "ADJUST THESE VALUES" part to experiment and improve your solutions.

Optionally, you can also improve your algorithm by implementing/changing strategies used

[45]:
```
# Grading (bonus points):
# 0.4 pts bonus for the optional improvement: if you implement different␣
 ↪strategy, it works and produces significantly better results
# Different strategy could be a different implementation of the mutation/
 ↪crossover/parent or survival selection mechanisms


# Implement your strategy here and integrate with the Evolutionary Algorithm
# PLEASE FILL IN

###################
```

[46]:
```
######################################################################

population_size = [  # ADJUST THESE VALUES
    50,
    50,
    50,
    50,
]
p_crossover = [  # ADJUST THESE VALUES
    0.9,
    0.9,
    0.9,
    0.9,
]
m_rate = [  # ADJUST THESE VALUES
    0.1,
    0.9,
    0.3,
```

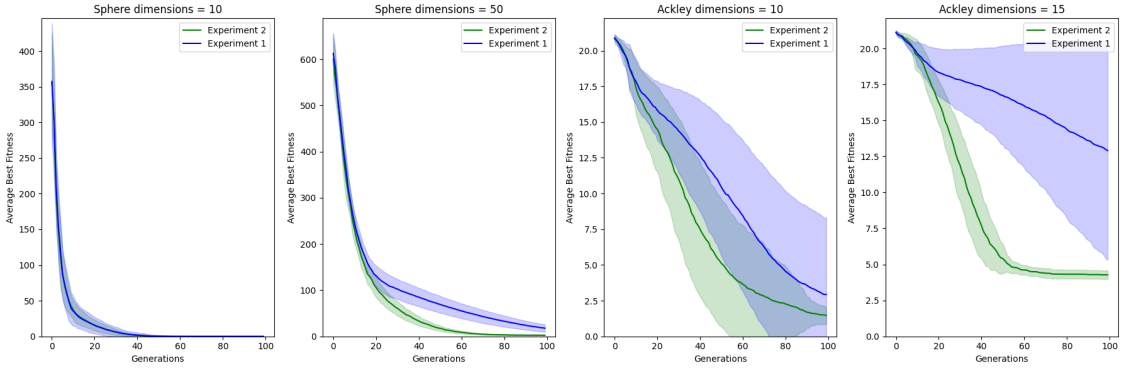
```
    0.9,
]


# Remove the line above once you've made the changes you want

#####################################################################
```

Running the experiment again

[47]:
```
avgs_experiment_2, stds_experiment_2, all_runs_experiment_2 = run_experiment(
    population_size, p_crossover, m_rate

)
```

Plotting the new and previous results

[48]:
```
labels = [
    "Sphere dimensions = 10",
    "Sphere dimensions = 50",
    "Ackley dimensions = 10",
    "Ackley dimensions = 15",
]  # DO NOT CHANGE


generate_subplot_function(
    avgs_experiment_1,
    stds_experiment_1,
    labels,
    avgs_experiment_2=avgs_experiment_2,
    stds_experiment_2=stds_experiment_2,
    n_columns=4,
    n_queens=None,
)
```



Measuring the statistical significance of differences of two sets of experiment results

The final step is to calculate whether the results from your set of hyperparameters (experiment 2) resulted in significantly better results.

To do this, we will perform a statistical test know as rank-sum test.

You will calculate the rank-sum for the results of the first and second experiments.

[49]:
```
alpha = 0.05

# this loops over the 4 different functions we have
#   (Sphere dimensions = 10, Sphere dimensions =50, Ackley dimensions = 10,␣
 ↪Ackley dimensions = 15)
labels = [
    "Sphere function 10 dimensions:",
    "Sphere function 50 dimensions:",
    "Ackley function 10 dimensions:",
    "Ackley function 15 dimensions:",
]
for i in range(4):
    runs_exp_1 = all_runs_experiment_1[i]
    runs_exp_2 = all_runs_experiment_2[i]

    best_per_run_exp_1 = [sublist[-1] for sublist in runs_exp_1]
    best_per_run_exp_2 = [sublist[-1] for sublist in runs_exp_2]

    t_statistic, p_value = ranksums(best_per_run_exp_1, best_per_run_exp_2)
    if p_value < alpha:
        # if np.mean(best_per_run_exp_1) < np.mean(best_per_run_exp_2):
        print(
            labels[i],
            "Experiment 1 average:",
            np.mean(best_per_run_exp_1),
            ", Experiment 2 average:",
            np.mean(best_per_run_exp_2),
            ",significant difference.",
        )
        # else:
        #    print(labels[i], "significant difference. Experiment 2 is better.")
    else:
        # if np.mean(best_per_run_exp_1) < np.mean(best_per_run_exp_2):
        print(
            labels[i],
            "Experiment 1 average:",
            np.mean(best_per_run_exp_1),
            ", Experiment 2 average:",
            np.mean(best_per_run_exp_2),
            ",no significant difference.",
```

```
        )
        # else:
        #    print(labels[i], "no significant difference. Experiment 2 is␣
↪better.")
```

```
Sphere function 10 dimensions: Experiment 1 average: 0.002718504908408991 ,
Experiment 2 average: 0.0011830872724541194 ,significant difference.
Sphere function 50 dimensions: Experiment 1 average: 17.785292309767982 ,
Experiment 2 average: 2.51770404165042 ,significant difference.
Ackley function 10 dimensions: Experiment 1 average: 2.931441329504097 ,
Experiment 2 average: 1.4617611282653442 ,significant difference.
Ackley function 15 dimensions: Experiment 1 average: 12.905017917776291 ,
Experiment 2 average: 4.280965982017202 ,significant difference.
```

Question 1.4 (0-0.3-0.6-0.9-1.2 pt): Please improve the results significantly in each case (i.e. on Sphere and Ackley functions with 10, 50 and 10, 15 dimensions respectively) and discuss what kinds of changes you had to do to achieve this improvement. Please provide your reasoning why the new parameter settings worked better.

Grading:

0 pts: No answer or non of the cases were improved.

0.3 pts: At least one of the cases improved significantly but reasons why were not clearly explained.

0.6 pts: One or two cases improved significantly and the reasons why were somewhat clear.

0.9 pts: Three to four cases improved significanlty and the reasons why are clear.

1.2 pts: All of the cases improved significanlty and the reasons why are clear.

Answer:

For all four cases, I kept the population size constant. The purpose of this experiment was to
mutation rate and crossover probability- and to understand the possibility of enhancing the opt

Sphere function 10 dimensions: (Experiment 1 average: 0.002718504908408991 , Experiment 2 average: 0.0018459482023169) ,significant difference.

I kept the low mutation rate because the Sphere function is unimodal and has simpler optimisati

The significant difference for better solution is provided by increasing the crossover probabil

Sphere function 50 dimensions: (Experiment 1 average: 17.785292309767982 , Experiment 2 average: 2.452626965047309) ,significant difference.

Despite the Sphere function being unimodal and not complex, the increased dimensions challenge

Ackley function 10 dimensions: (Experiment 1 average: 2.931441329504097 , Experiment 2 average: 1.5751166406027985) ,significant difference.

Compared to the Sphere function, the Ackley function has numerous local minima and is a much m

Ackley function 15 dimensions: (Experiment 1 average: 12.905017917776291 , Experiment 2 average: 4.291578453772402) ,significant difference.

```
Since Ackley is a complex function, and dimensions have increased, it became more challenging
```

1.2 Part 2: Discrete Optimization (The N-Queens Problem, 5 points total)

Implement an Evolutionary Algorithm for the **n-queens problem**. Below is a visualization of a solution for the n-queens problem when n = 4. Observe that none of the queens are attacking each other.

We would like to implement an EA that can find a solution for any given N by N board but in this case it is required to place N queens where none of them attack each other.

You may use the implementation of the EA you used for solving continuous problems in Part 1. But remember, this is a discere problem so you would need to think about how to represent the solutions and search using the evolutionary operators.

Consider, what changes you would need to do!

It is usually better to start simple and generalize your implementation. So, let's start with the case when $\mathbf{N = 4}$.

Question 2.1 (0-0.3 pt): How do you represent a solution (a 4 queen placement on a 4x4 chess board)? In particular, specify the length of your genotype representation, what each gene (dimension) represents, and what values they can get.

Answer:

By N-Queens problem, there can be only one queen present per row, column and diagonal. There are many ways to represent the solution. I represented it with an array; each index is for a row and the elements in it for the column. Imagine a chessboard with on the top a, b, c, d; which will be representing the columns. And on the left side e, f, g, h under each other; representing the rows.

For instance, by 4x4 chessboard; the genotype representations aka the solution could be an array with four elements. In this case each element can get values from 1 to 4. The lengtht of the solution would be then 4 and each gene(or dimension as mentioned) represents the column which will be the queen is placed for rows 1 to 4.

Question 2.2 (0-0.2 pt): Please write down an example representation and discuss what it means.

Answer: Lets assume we have 4x4 chessboard. the representation of the solution could be: [2, 4, 1, 3]. This solution ensures that each queen has an unique place on the board. Which means; - The queen in the first row, is in the second column. - The queen in the second row, is in the fourth column. - The queen in the third row, is in the first column. - The queen in the fourth row, is in the third column.

```
   a b c d
e  . Q . .
f  . . . Q
```

```
g  Q . . .
h  . . Q .
```

Question 2.3 (0-0.2 pt): How many possible solutions can be generated in your representation?

Answer:

Amount of possible solutions are depends on the permutation of N elements. In my example: N=4 so there are 4 factorial = 24 possible solution. However, this does not mean that all the solutions are valid, since there are conditions in n-queen problem that has to be required. So there is a great chance that there will be less than 24 possible solutions, considering for N=4 there are only 2 valid solutions. But ofcourse the solutions are depends on the amount of N (aka dimensions and number of queens).

Task 2.1 (0-0.20-0.40-0.80): Implementation of solution encoding, visualization and evaluation functions.

[50]:
```
####################################################################

example_solution =  [2, 4, 1, 3]

####################################################################
```

Write a function below that can visualize your solution. For instance, the output may look like below, a matrix representing the 4x4 chess board where each Q indicates a queen placement and dots are empty cells.

```
. Q . .
. . . Q
Q . . .
. . Q .
```

[51]:
```
def visualize_solution(solution):
    """Visualize the placement of queens on the chessboard."""

    ####################################################################
    L = len(solution)
    for i in range(L):
        row = ['. '] * L
        row[solution[i] - 1] = 'Q '
        print(''.join(row))

    ####################################################################
```

Write the evaluation function to assess how good your solution is.

[52]:
```
def evaluate_solution_n_queens(solution):
    """Calculate the fitness of a solution based on the number of non-attacking␣
 ↪queens."""
    n = len(solution)
    fitness = 0

    # Calculate fitness as the number of non-attacking queens
    for i in range(n):
        is_attacked = False
        for j in range(n):
            if i != j:
                # Check if the queens attack each other, horizontally or on␣
 ↪diagonal
                if solution[i] == solution[j] or abs(solution[i] - solution[j])␣
 ↪== abs(i - j):
                    is_attacked = True
                    break
        if not is_attacked:
            fitness += 1  # INcrease fitness score for each safe queen
    return fitness
```

Try your implementations to see if your solution encoding matches to visualization and whether the fitness is computed correctly.

[53]:
```
##################################
# Grading
# 0 pts: No attempt, representation discussed does not match with the␣
 ↪implementation and visualization, fitness is not correct.
# 0.20 : Solution representation matches with visualization, fitness␣
 ↪computation is not correct, no explanation in the code.
# 0.50 : Solution representation matches with visualization, fitness␣
 ↪computation is correct, no explanation in the code.
# 0.80 : Solution representation matches with visualization, fitness␣
 ↪computation is correct, the implementation explained well.
#####################################################################


###################### DO NOT CHANGE ##########################
print("Genotype (solution representation):", example_solution)
print("Phenotype (solution visualization):")
visualize_solution(example_solution)
print("Solution fitness", evaluate_solution_n_queens(example_solution))
###################### DO NOT CHANGE ##########################
```

```
Genotype (solution representation): [2, 4, 1, 3]
Phenotype (solution visualization):
```

```
. Q . .
. . . Q
Q . . .
. . Q .
Solution fitness 4
```

Task 2.2 (0-0.4-0.8-1.2-1.6 pt): Write an evolutionary algorithm that can initialize a population of solutions and finds N queen placement to NxN board optimizing the number of attacks (could be minimization or maximization based on your evaluation function of the solutions).

[163]:
```
##################################
# Grading
# 0 pts if the code does not work, code works but it is fundamentally incorrect
# 0.4 pts if the code works but some functions are incorrect and it is badly␣
 ↪explained
# 0.8 pts if the code works but some functions are incorrect but it is␣
 ↪explained well
# 1.2 pts if the code works very well aligned with the task without any␣
 ↪mistakes, but it is badly explained
# 1.6 pts if the code works very well aligned with the task without any␣
 ↪mistakes, and it is well explained
#####################################################################


#initialize a population of solutions for the N queens problem where num_dims =␣
 ↪N
def initialization_n_queens(population_size, num_of_dims):
    """Generate a population of solutions."""
    ####################################################################

    # PLEASE FILL IN

    x = [np.random.permutation(num_of_dims).tolist() for _ in␣
 ↪range(population_size)] #Randomly permute a sequence through iterating over␣
 ↪the population


   ####################################################################

    return x #return population


def evaluation_n_queens(x):
    """Evaluate the whole population and return the fitness of each."""

    return [evaluate_solution_n_queens(solution) for solution in x]
```

```
def crossover_n_queens(x_parents, p_crossover):
    """Perform crossover to create offsprings."""

    ######################################################################

    # PLEASE FILL IN
    # offspring = ?

    offspring = []

    for i in range(0, len(x_parents) - 1, 2): #this takes pairs of parents for␣
 ↪each iteration
        parent1 = x_parents[i]
        parent2 = x_parents[i+1]

        if np.random.rand() < p_crossover: #using crossover probability to make␣
 ↪offsprings.

            size = len(parent1)
            child1 = [None]*size
            child2 = [None]*size
            crosspoint1, crosspoint2 = np.sort(np.random.choice(range(size), 2,␣
 ↪replace=False)) #Randomly seelcting the crossover points

            #copying the segment at the crossover points for offsprings, OX␣
 ↪method
            child1[crosspoint1:crosspoint2] = parent1[crosspoint1:crosspoint2]
            child2[crosspoint1:crosspoint2] = parent2[crosspoint1:crosspoint2]

            # Loop through the parent lists to fill None parts in child lists␣
 ↪with the unique queens.
            # PMX method; uniquely placing the remaining elements from each␣
 ↪parent into the offspring
            for cho in range(size):
                if parent2[cho] not in child1:
                    child1[child1.index(None)] = parent2[cho]
                if parent1[cho] not in child2:
                    child2[child2.index(None)] = parent1[cho]

            offspring.extend([child1, child2])

        else:
            offspring.extend([parent1, parent2])
```

```
    ######################################################################

    return offspring

def mutation_n_queens(x, mutation_rate):
    """Apply mutation to an individual."""
    for i in range(len(x)):
        if np.random.rand() < mutation_rate: #mutation rate to have the change␣
 ↪of mutation
            # Perform mutation: swap mutation
            idx1, idx2 = np.random.choice(len(x[i]), 2, replace=False)
            x[i][idx1], x[i][idx2] = x[i][idx2], x[i][idx1]
    return x


def parent_selection_n_queens(x, f):
    """Select parents for the next generation using roulette wheel selection."""
    x_parents = []
    f_parents = []
    population_size = len(x)
    tournament_size = 2 # The number of individuals to compete in each␣
 ↪tournament

    """Tournament Selection"""
    for _ in range(population_size):
        # Randomly select tournament_size individuals from the population
        indices = np.random.choice(population_size, tournament_size,␣
 ↪replace=False) #Gives two indices [a, b] for tournament
        #x[i] = [genes], f[i] = fitness score of that array
        tournament_individuals = [(x[i], f[i]) for i in indices] #Contains two␣
 ↪arrays on the given indices(each time randomly)

        # Select the best individual from the tournament
        #the best fitness value for a population is the greatest fitness value␣
 ↪for any individual in the population.
        winner = max(tournament_individuals, key=lambda individual:␣
 ↪individual[1])

        # Appending the winner to the parent list
        x_parents.append(winner[0])
        f_parents.append(winner[1])


    return x_parents, f_parents
```

```
def survivor_selection_n_queens(x, f, x_offspring, f_offspring):
    """Select the survivors, for the population of the next generation"""


    ######################################################################

    # PLEASE FILL IN
    # x = ?
    # f = ?
    """truncation selection method (form of elitism ), where only the best␣
 ↪individuals with the best fitness survive to the next generation."""
    # Combine the current population and offspring into one extended pool
    combined_population = x + x_offspring
    combined_fitness = f + f_offspring

    # Determine the number of individuals to retain (assuming population size␣
 ↪remains constant)
    population_size = len(x)

    # Get the indices of the individuals with the highest fitness scores
    # Sort by fitness ascendingly for maximization
    sorted_indices = sorted(range(len(combined_fitness)), key=lambda i:␣
 ↪combined_fitness[i], reverse=True)

    # Clear the original population lists
    x.clear()
    f.clear()
    # Fill the original lists with the top individuals based on sorted indices
    for index in sorted_indices[:population_size]:
        x.append(combined_population[index])
        f.append(combined_fitness[index])
    ######################################################################

    return x, f

def ea_n_queens(population_size, max_fit_evals, p_crossover, m_rate,␣
 ↪num_of_dims):
    # Calculate the maximum number of generations
    max_generations = int(max_fit_evals / population_size)

    ##########################
    #  PLEASE FILL IN
    x = initialization_n_queens(population_size, num_of_dims)
    f = evaluation_n_queens(x)

    ##########################
```

```
    # Get best individual and append to list
    idx = np.argmax(f)
    x0_best = x[idx]
    f0_best = f[idx]
    x_best = [x0_best]
    f_best = [f0_best]

    # Loop over the generations
    for _ in range(max_generations - 1):
        # Select population size parents

        ##########################
         #PLEASE FILL IN
        x_parents, f_parents = parent_selection_n_queens(x, f)
        # Crossover
        x_offspring = crossover_n_queens(x_parents, p_crossover)
        # Mutation
        x_offspring = mutation_n_queens(x_offspring, m_rate)
        # Evaluate offspring
        f_offspring = evaluation_n_queens(x_offspring)
        # Survivor selection
        x, f = survivor_selection_n_queens(x, f, x_offspring, f_offspring)

        ##########################

        # Find the best individual in current generation and add to the list
        idx = np.argmax(f)
        xi_best = x[idx]
        fi_best = f[idx]
        if fi_best > f_best[-1]:
            x_best.append(xi_best)
            f_best.append(fi_best)
        else:
            x_best.append(x_best[-1])
            f_best.append(f_best[-1])

        # Append the best individual to the list
        f_best.append(fi_best)
        x_best.append(xi_best)

    return x_best, f_best
```

Results:

Run the code below to run an EA for N=8, 16 and 32, and visualize the best solutions found.

[207]:
```
print("Case when N=8:")


x_best, f_best = ea_n_queens(100, 1000, 0.6, 0.1, 8)



print("Best fitness:", f_best[-1])
print("Best solution found:")
visualize_solution(x_best[-1])
```

```
Case when N=8:
Best fitness: 8
Best solution found:
. . . . Q . . .
. . . . . . . Q
. . . Q . . . .
Q . . . . . . .
. . . . . . Q .
. Q . . . . . .
. . . . . Q . .
. . Q . . . . .
```

[238]:
```
print("Case when N=16:")
x_best, f_best = ea_n_queens(200, 10000, 0.9, 0.1, 16)


print("Best fitness:", f_best[-1])
print("Best solution found:")


visualize_solution(x_best[-1])
```

```
Case when N=16:
Best fitness: 16
Best solution found:
. . Q . . . . . . . . . . . . . .
. . . . . . . . Q . . . . . . . .
. . . Q . . . . . . . . . . . . .
. . . . . . . . . . . . . Q . . .
. . . . Q . . . . . . . . . . . .
. . . . . . . . . . . . Q . . . .
Q . . . . . . . . . . . . . . . .
. . . . . Q . . . . . . . . . . .
. . . . . . . . . . . . . . Q . .
. Q . . . . . . . . . . . . . . .
. . . . . . . Q . . . . . . . . .
. . . . . . . . . . . . . . . Q .
```

```
. . . . . . . . . . Q . . . . .
. . . . . . . . . . . . . . . Q
. . . . . . . . . Q . . . . . .
. . . . . Q . . . . . . . . . .
```

[110]:
```python
print("Case when N=32:")
x_best, f_best = ea_n_queens(100, 10000, 0.9, 0.3, 32)

print("Best fitness:", f_best[-1])
print("Best solution found:")
visualize_solution(x_best[-1])
```

```
Case when N=32:
Best fitness: 32
Best solution found:
. . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . .
. Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . .
. . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . .
Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . .
. . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . .
. . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q . .
. . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . . .
. . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . Q . . . .
. . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q
. . . . . . . . . . Q . . . . . . . . . . . . . . . . . . . . .
. . . . Q . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Q .
. . . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . .
. . . . . . . . Q . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . Q . . . . . . . . . . .
. . . . . . . . . . . . Q . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . Q . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . .
. . . . . . . . . . . . . . . . . . Q . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . Q . . . . . . .
. . . . . . Q . . . . . . . . . . . . . . . . . . . . . . . . .
. . Q . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

Question 2.4 (0-0.2-0.4-0.6-1-1.5 pt): Describe the results. What was the fitness found for each case? Were you able to find fitness scores of 8, 16 and 32 for N=8, 16 and 32 cases? How did you find them? Did you try improving the results by testing different parameters and/or evolutionary operators?

Grading:

0 pts: no solution or visualization provided.

0.2 pts: Solutions and visualizations were provided, no insights provided how the results achieved.

0.4 pts: Solutions and visualizations were provided, at least for N=8, a optimum solution was found, limited/no insights provided on how this result is achieved.

0.6 pts: Solutions and visualizations were provided, at least for N=8, a optimum solution was found, insights on how this result is achieved provided.

1 pts: Solutions and visualizations were provided, for N=8 and N=16, optimum solutions were found, insights on how this result is achieved provided.

1.5 pts: Solutions and visualizations were provided, for N=8, N=16 and N=32, optimum solutions were found, insights on how this result is achieved provided.

Answer:

I was able to find fitness scores of 8, 16, and 32 for N = 8, N = 16, and N = 32, respectively.

This was a maximization problem, so the higher the fitness score, the better it was. So, I aimed to find the best fitness score as the highest. Since, in this case, there weren't any objective functions, I designed the **evaluation function** to calculate the fitness for the solution while keeping track of the non-attacking queens. Thus, the fitness score is calculated for each non-attacking queen. In this approach, the function was able to detect the queens that weren't on the same diagonal, row, or column.

The initialization of the queens is done with arrays in which indices represent the rows, and the values of those indices represent the columns. Additionally, the function initializes a population with unique rows and columns for each queen by using permutations. For the parent and survival selection, I kept the functions from part 1; the only changes that I made in the functions were to change "min" to "max" so they would work for the maximization problem. Considering the tournament selection and elitism in the survival selection functions, the algorithm was able to work efficiently.

For the **crossover function**, my method was a hybrid one. I combined OX (Ordered Crossover) and PMX (Partially Mapped Crossover) to get the optimal solution. OX is a crossover method that allows the function to select two parent chromosomes, and a segment of genetic material (genes) from one parent is inserted into the offspring in the same order it appears in the parent adn the remaining genes are filled in using the genes from the second parent. I chose to use OX and PMX techniques because the N-queen problem is a permutation-based problem for which OX and PMX are generally used. - Initially, I tried to use only OX but encountered some NoneType errors or problems later on in the mutation function. I had to come up with a solution such as PMX, which ensures that the offspring retains some of the characteristics of both parents while still generating new solutions and thus fully fills the list to prevent the NoneType error.

For the mutation part, I used the basic swapping method, which is performed by the mutation rate. However, to find the optimal solution for N=8, N = 16 and N = 32, improved the results by changing the parameters.

- For N= 8, I reduced the population size and increased the crossover probability. N=8 is not a complex problem, so optimizer does not have to search through numerious variations. Therefore, generating and having huge genetic diversity of the population throught the huge population size seemed to cause challenges for the performance of the optimizer. Thus at first it did not provided correct solution even though the fitness was optimum. To prevent that, i kept the population size lower and increased crossover population only by 0.1 to keep the balance.
- For N = 16, I reduced the population size, mutation rate to prevent unnecessary gene variations that could lead to a decrease in the performance of the optimizer. Also, with the reduction in population size, the algorithm performed faster, and without the unnecessary variations, the solution was provided.
- For N = 32, the situation was different. To find the optimal solution for N = 32, the optimizer should have more variations of the genes so that a solution for the problem could be provided with the best fitness score. So, I increased the probability of crossover and mutation rate to increase the chance of crossover, thus leading to more variations that could lead to the solution. This led the algorithm to find the best fitness for N = 32.

Thus with this prametric changed the optimal fitness along with the solutions provided.

Plotting the average performance of the algorithm Use the cells below to plot the results of your algorithm similar to Part 1. The plots should show average and std of 10 runs of EA for n-queens problem for N=8, 16, 32.

[208]:
```
def run_experiment_n_queens(population_size, p_crossover, m_rate):
    # These are the hyperparameters of your evolutionary algorithm. You are not␣
 ↪allowed to change them.

    max_fit_evals = 10000

    fitness_8 = []
    fitness_16 = []
    fitness_32 = []

    runs = 10

    for run in range(runs):
        print("Run: ", run)

        _, f_best_8 = ea_n_queens(
            population_size[0], max_fit_evals, p_crossover[0], m_rate[0],␣
 ↪num_of_dims=8
        )
```

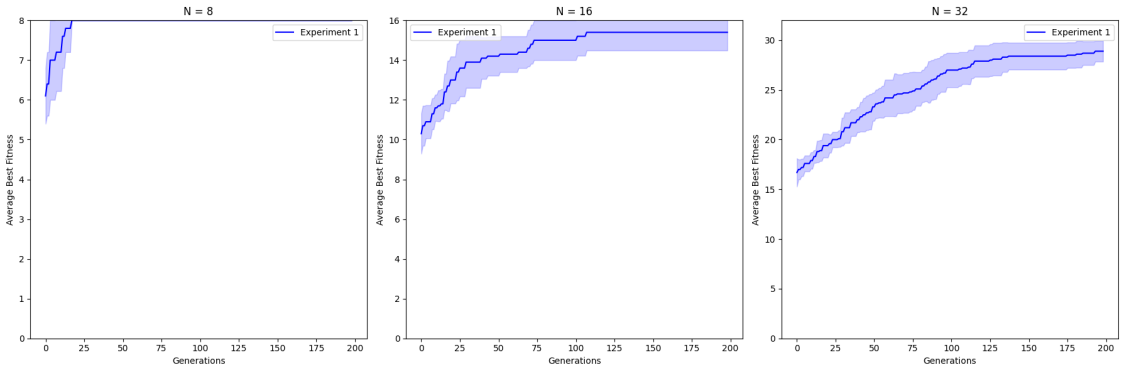
```
        _, f_best_16 = ea_n_queens(
            population_size[1], max_fit_evals, p_crossover[1], m_rate[1],␣
 ↪num_of_dims=16
        )
        _, f_best_32 = ea_n_queens(
            population_size[2], max_fit_evals, p_crossover[2], m_rate[2],␣
 ↪num_of_dims=32
        )
        fitness_8.append(f_best_8)
        fitness_16.append(f_best_16)
        fitness_32.append(f_best_32)

    avg_8, std_8 = calculate_mean_std(fitness_8)
    avg_16, std_16 = calculate_mean_std(fitness_16)
    avg_32, std_32 = calculate_mean_std(fitness_32)

    avgs = [avg_8, avg_16, avg_32]
    stds = [std_8, std_16, std_32]
    all_runs = [fitness_8, fitness_16, fitness_32]

    return avgs, stds, all_runs
```

[209]:
```
population_size = [100, 100, 100]  # not allowed to change
p_crossover = [0.8, 0.8, 0.8]  # not allowed to change
m_rate = [0.1, 0.1, 0.1]  # not allowed to change

avgs_experiment_1, stds_experiment_1, all_runs_experiment_1 =␣
 ↪run_experiment_n_queens(
    population_size, p_crossover, m_rate
)
```

```
Run:  0
Run:  1
Run:  2
Run:  3
Run:  4
Run:  5
Run:  6
Run:  7
Run:  8
Run:  9
```

[210]:
```
labels = ["N = 8", "N = 16", "N = 32"]

generate_subplot_function(
    avgs_experiment_1,
    stds_experiment_1,
```

```
    labels,
    avgs_experiment_2=None,
    stds_experiment_2=None,
    n_columns=3,
    n_queens=[8, 16, 32],
)
```



Question 2.5 (0-0.2-0.4 pt) Describe the average performance of the algorithm. What was the maximum average fitness found for each case? Do you see any differences between the problem cases?

Answer:

The average performance of the algorithm across the generations indicates a consistent improvements for all the cases. - For N=8 EA reaches the optimum solution within 50 generations and remain constant at the best average, which is quicker than the other cases considering their improvements continued within 0-115 or 0-200 generations. Additionally, as can be seen from the figures; N=8 indicates a simpler problem space that leads algorithm to explore more effectively and find the solution quicker. - For N=16, algorithm's improvement continued within approximately 115 generations and stayed stable after finding the solution. This mean that the search space of it could be more complex but still within the capability of algorithm to find the best fitness efficiently. - For N=32, since there was more queen to consider and the problem become more complex, finding the solution for the optimiser took more than 200 generations without any stabilisation. Additionally, it has more variance as indicated by the wider confidence interval. Thus the efficiency of the algorithm reduced due to the complex problem space.

From these observations it is true to say that the complexity of the search space increases if the size of the problem, this case N, increases. Therefore, reaching the best fitness possibly takes more generations for the algorithm. This is also due to the complexity and nature of the N-Queens problem that configurations grows exponentially with N and which definitely challenges the optimizer. The wider confidence interval in larger N values also indicates greater variability in the algorithm's performance, which could be attributed to the increased difficulty of finding non-attacking arrangements as the board size(N) grows.