AI LAB 2

תהליכים גנטיים

Kareen Ghattas - 207478728
Hakeem Abu shqara - 207691312

How to run the code:

1) Select Algorithm type: 'GA_Binary' to see the results and 'GA_Binary_vs_GA' to compare between the string and binary representation.

4) Select Algorithm type: select the algorithm you want to run, 'GA_NQueens_Mutation', 'GA_BinPacking_Mutation', 'GA_String_Mutation'

And then change these parameters as you want:

- **mutation_method:** a string that specifies the mutation method to be used, such as "CONSTANT", "NON_UNIFORM", "ADAPTIVE", "THM", and "SELF_ADAPTIVE".
- **mutation_prob:** a float that specifies the initial mutation probability for each gene in an individual's genome.
- **threshold:** a float that is used as a threshold value in some of the mutation methods.
- **stagnation_gen:** an integer that is used in the "THM" method to track the number of generations in which there has been no improvement in the best fitness of the population.
- **decay_rate:** a float that is used in the "NON_UNIFORM" method to control the rate at which the mutation probability decreases over time.

6) Select Algorithm type: 'BinPacking_niching' and then select the Niching method you want: 'FITNESS_SHARING', 'CROWDING', 'SPECIATION'. And change the parameters sharing_radius, alpha, crowding_factor, num_species as you want (more explanation about each parameter and value examples in sction 6).

Or select Algorithm type 'Compare_niching' to show the comparison between the nichings.

7) Select Algorithm type: 'Exaptation'

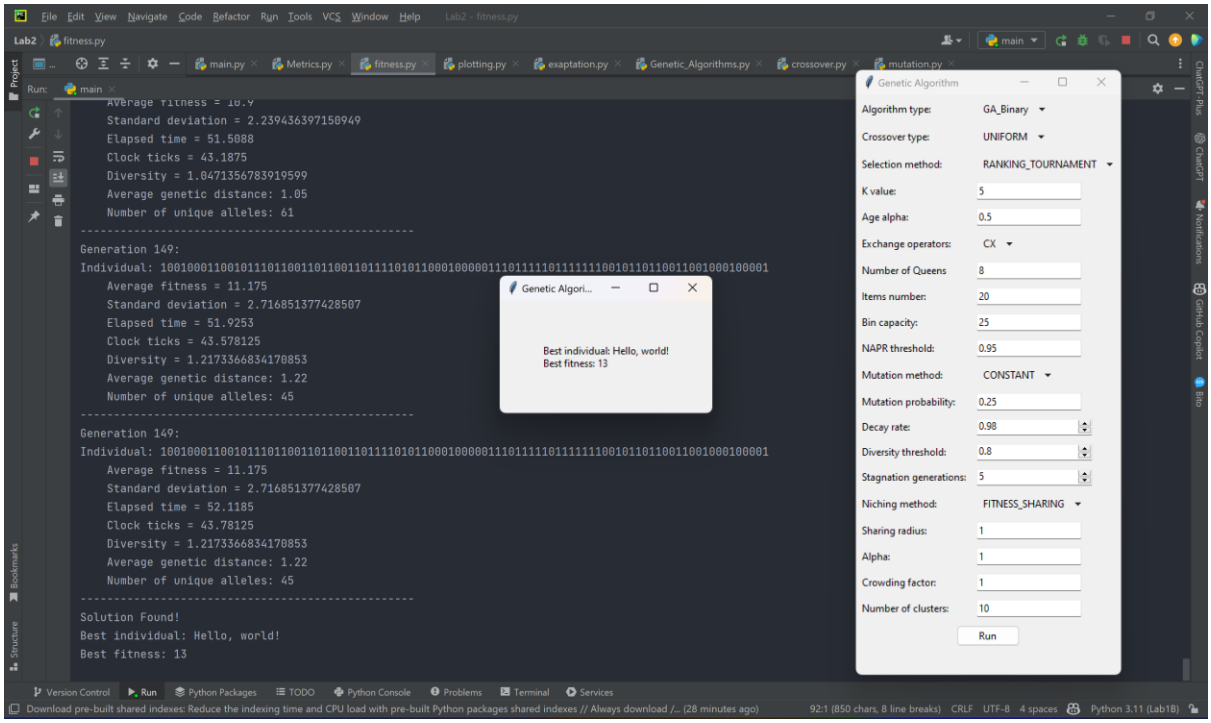
1. יצגו באופן בינארי את בעיית בול הפגיעה והריצו את האלגוריתם הגנטי כך שלמרחק בין אותיות תהיה חשיבות – השוו את התוצאה לתוצאות הקודמות שקיבלתם עבור אלגוריתם זה

To represent individuals in binary form, we modified the **generate_binary_individual** function to generate a list of binary strings using 7 bits to represent each character. We also modified the **crossover_binary** function to perform crossover on binary strings instead of characters. Finally, we modified the **binary_hit_stamp_fitness** function to calculate the fitness of individuals represented in binary form using the Levenshtein distance between the binary strings and the corresponding target character's binary string. The rest of the genetic algorithm remains the same as before.

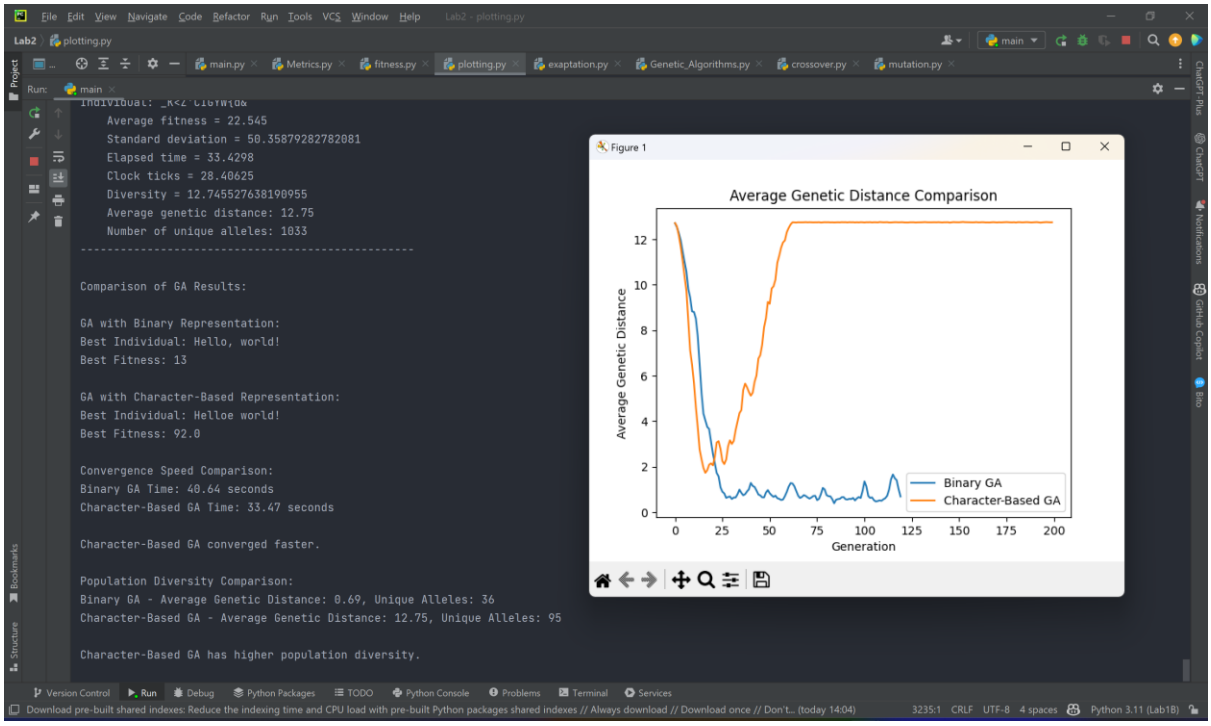
```
def levenshtein_distance(s, t):
    m, n = len(s), len(t)
    d = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(m + 1):
        d[i][0] = i
    for j in range(n + 1):
        d[0][j] = j
    for j in range(1, n + 1):
        for i in range(1, m + 1):
            if s[i - 1] == t[j - 1]:
                cost = 0
            else:
                cost = 1
            d[i][j] = min(d[i - 1][j] + 1, d[i][j - 1] + 1, d[i - 1][j - 1] + cost)
    return d[m][n]


def binary_hit_stamp_fitness(individual):
    distance = sum(levenshtein_distance(individual[i], format(ord(GA_TARGET[i]), '07b')) for i in range(len(GA_TARGET)))
    return len(GA_TARGET) - distance
```

This fitness function uses the Levenshtein distance algorithm to calculate the distance between the binary representation of the individual and the binary representation of the target string. The Levenshtein distance is a metric for measuring the difference between two strings. It is defined as the minimum number of insertions, deletions, or substitutions required to transform one string into the other.
In this function, the Levenshtein distance is calculated for each character in the target string by comparing it to the corresponding character in the binary representation of the individual. The sum of these distances is then subtracted from the length of the target string to obtain the fitness value. The fitness value represents how close the individual is to the target string, with a higher fitness value indicating a closer match.



We compared the original algorithm with the new one with the binary presentation:



Based on the graph, the population diversity of the binary GA remains relatively stable throughout the generations, with some small ups and downs, while the diversity of the character-based GA decreases significantly in the first generations and then increases again towards the end. This suggests that the binary GA is better at maintaining population diversity throughout the search process, while the character-based GA may struggle with maintaining diversity early on but eventually recovers towards the end.

2. הציגו את מטריקות הדימיון עבור כ"א מהבעיות שהמנוע שלכם פותר (מחרוזות – מרחק המינג וEDIT DISTANCE פרמוטציות – היפוכי קנדל טאו ואריזה – מטריקה שתבחרו)

** השוואה בין תצורות אריזה שונות יכולה להתבסס על כמות המכלים, הפיזור של העצמים בתוכם ויעילות האריזה בכל מיכל

String matching:

As we explained in the previous section, we added the Levenshtein distance. We also tried another kinds of metrics like **hamming_distance, jaccard_similarity, lcs_distance** that uses **longest_common_subsequence.**

We also tried to add them to our fitness function to see what is better for us, and we find that they all give a good results sometimes but the Levenshtein distance we see that is the best of them in most cases.

1. **Levenshtein distance:** Calculates the minimum number of single-character edits (insertions, deletions, or substitutions) required to change the candidate string into the target string.
2. **Hamming distance:** Calculates the number of positions at which the candidate string differs from the target string.
3. **Jaccard similarity:** This metric measures the similarity between two sets, in this case, the sets of binary digits (0s and 1s) present in the strings. It's a simple measure and doesn't account for the position or frequency of the digits.
4. **Longest Common Subsequence (LCS) distance:** This metric calculates the length of the longest common subsequence between two strings. It considers the order of characters but doesn't require characters to be contiguous.

* All their implementation are in the Metrics.py file, we didn't add them here cause we use the one provided earlier.

Bin Packing:

For the bin packing problem, we implemented the **kendall_tau_distance**:

Kendall's Tau distance is a measure of correlation between two rankings. In the bin packing problem, we use it to compare different packaging configurations by calculating the distance between the rankings of item assignments to the bins. A lower distance indicates a more similar distribution of items across the bins, while a higher distance indicates a more dissimilar distribution.

```python
# Bin packing metrics
def kendall_tau_distance(config1, config2):
    n = len(config1)
    inversions = 0

    for i in range(n):
        for j in range(i + 1, n):
            if (config1[i] - config1[j]) * (config2[i] - config2[j]) < 0:
                inversions += 1

    return inversions
```

N-Queens:

For the N-Queens problem, we implemented the **Non-Attacking Pairs Ratio** that measures the proportion of non-attacking queen pairs in the population.

Non-Attacking Pairs Ratio:

This metric focuses on the quality of the solutions in the population. It tells us the proportion of non-attacking queen pairs in the entire population. A higher ratio indicates that the algorithm is effectively finding and refining solutions with fewer attacking queen pairs. This metric helps us evaluate the performance of the algorithm in minimizing the attacking pairs of queens, which is the primary objective in the N-Queens problem.

```python
# N-Queens metrics
def non_attacking_pairs_ratio(population):
    total_pairs = 0
    non_attacking_pairs = 0

    for individual in population:
        size = len(individual)
        pairs = size * (size - 1) // 2
        total_pairs += pairs
        conflicts = n_queens_fitness(individual)
        non_attacking_pairs += (pairs - conflicts)

    ratio = non_attacking_pairs / total_pairs
    return ratio
```

We also implemented the **average_kendall_tau_distance** that's measure the average dissimilarity between the individuals in the population. A higher average Kendall Tau distance would indicate a more diverse population.

```
def average_kendall_tau_distance(population):
    total_distance = 0
    num_individuals = len(population)
    num_comparisons = 0

    for i in range(num_individuals):
        for j in range(i + 1, num_individuals):
            distance = kendall_tau_distance(population[i], population[j])
            total_distance += distance
            num_comparisons += 1

    average_distance = total_distance / num_comparisons
    return average_distance
```

Average Kendall Tau Distance:

This metric measures the diversity of the population. It calculates the average dissimilarity between the individuals in the population, which can give us insights into the exploration of the search space by the GA. A higher average Kendall Tau distance would indicate a more diverse population. This metric helps us assess the balance between exploration (searching for new solutions) and exploitation (refining existing solutions) in the GA.

3. הוסיפו למנוע שלכם תמיכה במנגנונים השונים של הפעלת מוטציות כפי שנלמדו

For this section we added **mutation.py** file that include these implementations:

```python
def update_mutation_probability(method, generation, mutation_prob, population, fitnesses, fitness_func, threshold=None,
                                stagnation_gen=None, decay_rate=None, fitness_func_args={}):
    if method == "CONSTANT":
        return mutation_prob

    elif method == "NON_UNIFORM":
        if decay_rate is None:
            decay_rate = 0.99
        return mutation_prob * (decay_rate ** generation)

    elif method == "ADAPTIVE":
        avg_fitness = sum(fitnesses) / len(fitnesses)
        if avg_fitness > threshold:
            return mutation_prob * 1.1
        else:
            return mutation_prob * 0.9

    elif method == "THM":
        if generation >= 1:
            prev_best_fitness = max(fitnesses)
            current_best_fitness = max([fitness_func(individual, **fitness_func_args) for individual in population])
            # current_best_fitness = max([fitness_func(individual) for individual in population])
            if abs(prev_best_fitness - current_best_fitness) < threshold:
                stagnation_gen += 1
                if stagnation_gen >= 5:
                    return mutation_prob * 2
            else:
                stagnation_gen = 0
        return mutation_prob

    elif method == "SELF_ADAPTIVE":
        mutation_probs = []
        for individual, fitness in zip(population, fitnesses):
            if fitness < threshold:
                mutation_probs.append(mutation_prob * 1.5)
            else:
                mutation_probs.append(mutation_prob)
        new_mutation_rate = sum(mutation_probs) / len(mutation_probs)
        return new_mutation_rate
```

```python
# Update mutation rate based on the selected method
if not constant_mutation_rate:
    mutation_rate = update_mutation_probability(mutation_method, generation, mutation_rate, population,
                                                fitnesses, fitness_func, threshold, stagnation_gen, decay_rate)
```

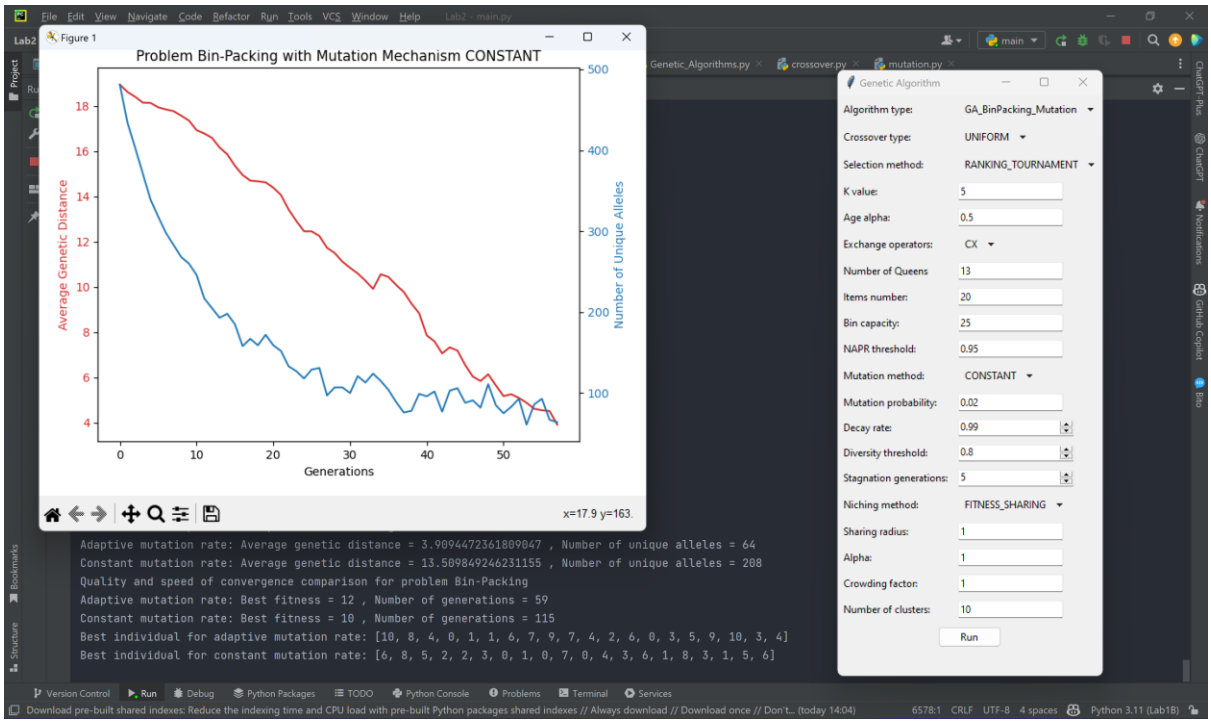
1. **CONSTANT**: This method keeps the mutation probability constant throughout the generations.
2. **NON_UNIFORM**: This method reduces the mutation probability by a certain decay rate at every generation.
3. **ADAPTIVE**: This method adjusts the mutation probability based on the average fitness of the population. If the average fitness is above a certain threshold, the mutation probability is increased by 10%, otherwise, it is decreased by 10%.
4. **THM**: This method increases the mutation probability if the population fitness has not improved beyond a certain threshold for a specified number of generations. The stagnation generation count is reset to zero if the fitness improves beyond the threshold.
5. **SELF_ADAPTIVE**: This method adjusts the mutation probability for each individual in the population based on their fitness. If an individual's fitness is below a certain threshold, their mutation probability is increased by 50%, otherwise, it remains the same. The new mutation probability is calculated as the average of all the individual mutation probabilities.

4. השוו בין הביצועים המקוריים של האלגוריתמים על שלושת בעיות הדוגמא אל מול מנגנוני המוטציה השונים וכן בין שיטות המוטציה עצמן:
   a. באמצעות מדידות של הגוון הגנטי
   b. באיכות ומהירות ההתכנסות לפתרון

Bin packing:

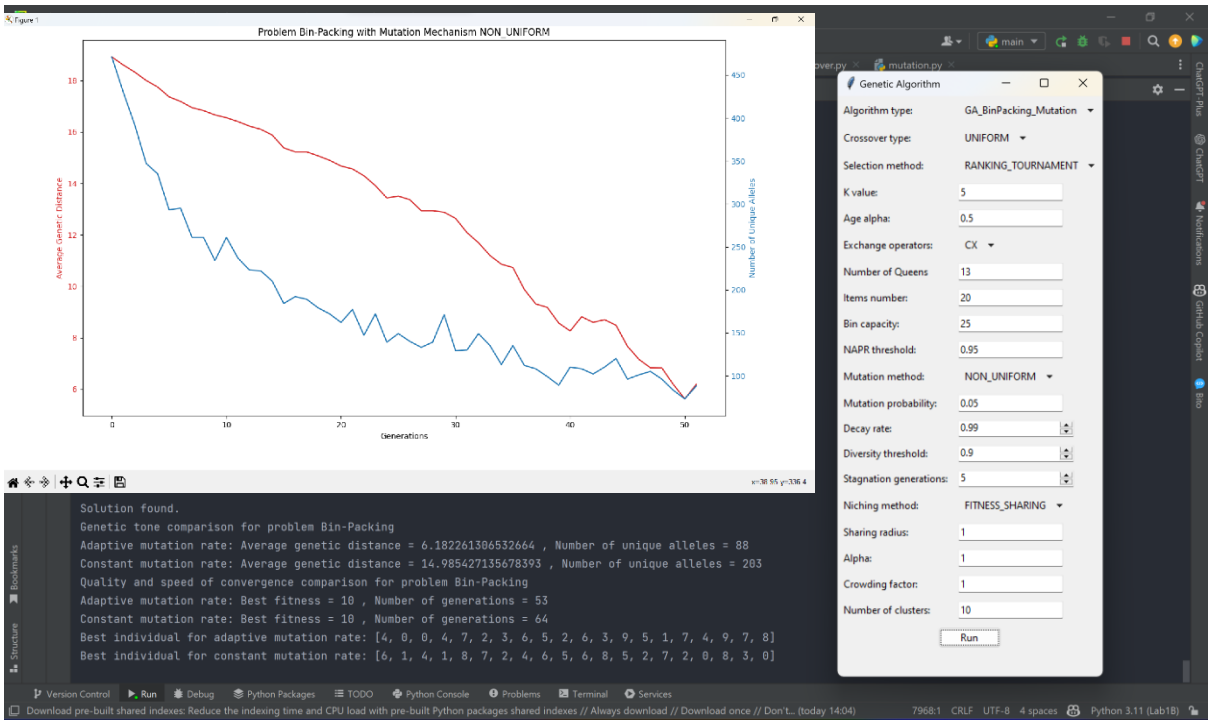
Mutation method: **"CONSTANT"**

- Mutation probability: 0.02



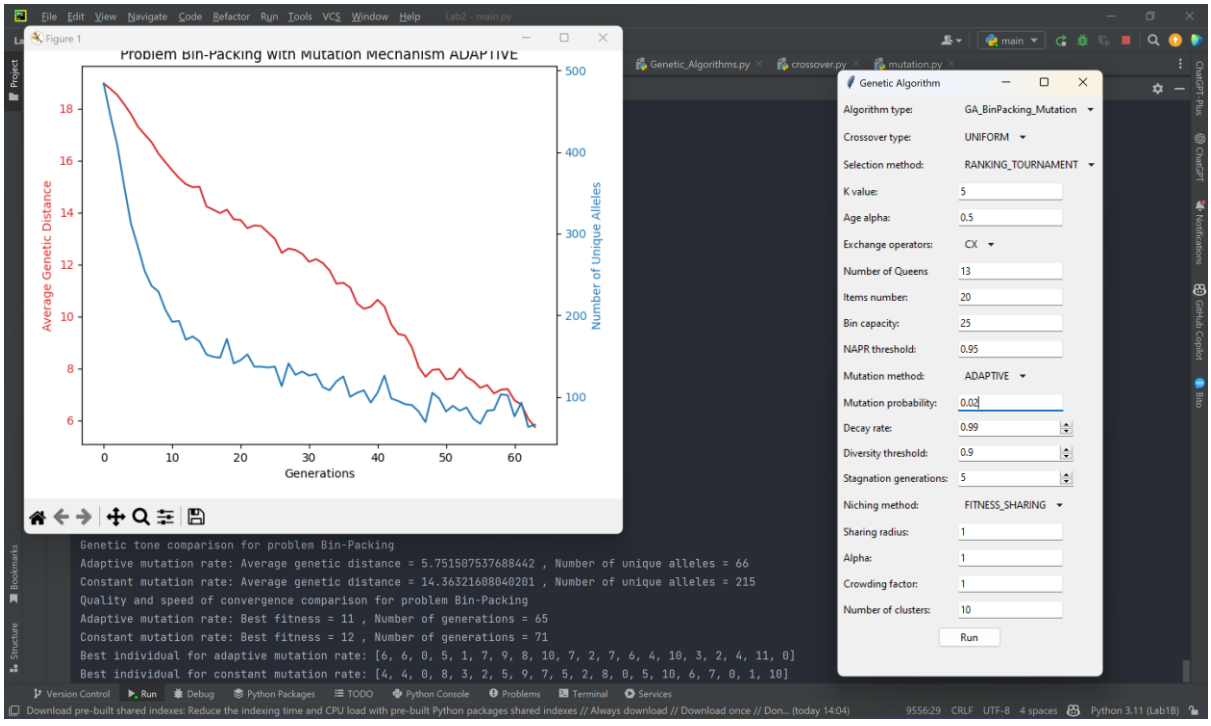
Mutation method: **"NON_UNIFORM"**

- Mutation probability: 0.05
- Decay rate: 0.99

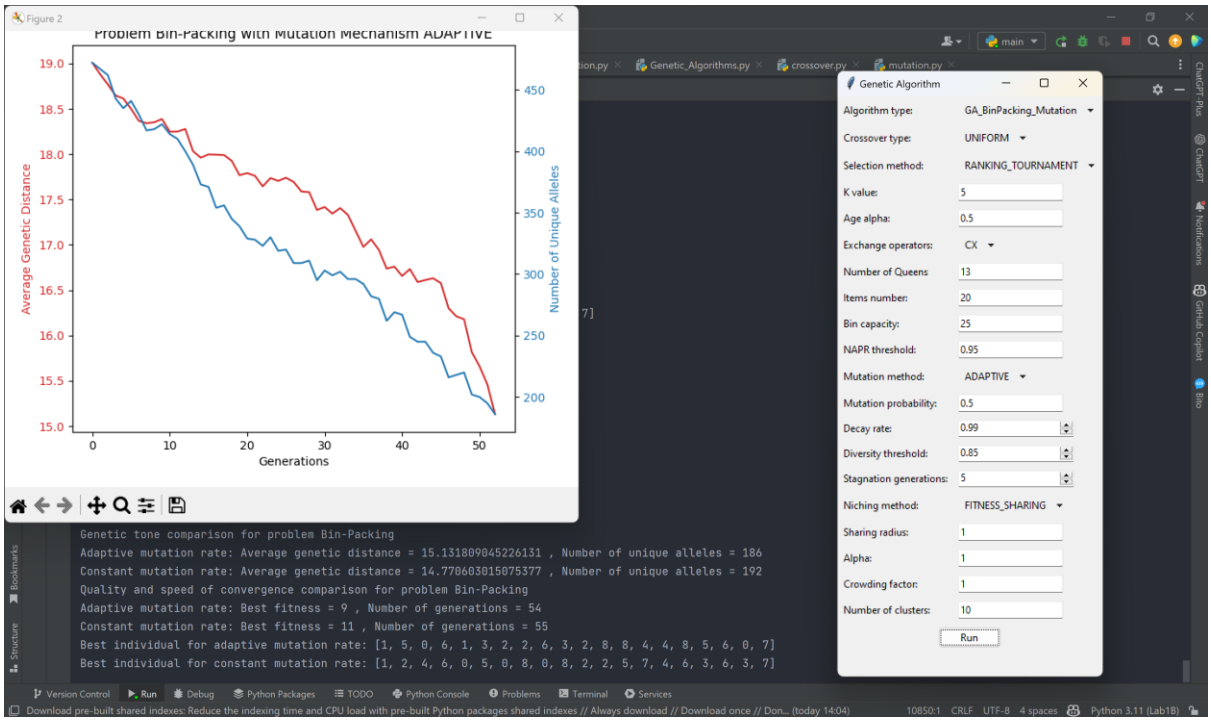


Mutation method: **"ADAPTIVE"**

- Mutation probability: 0.02
- Threshold: 0.99

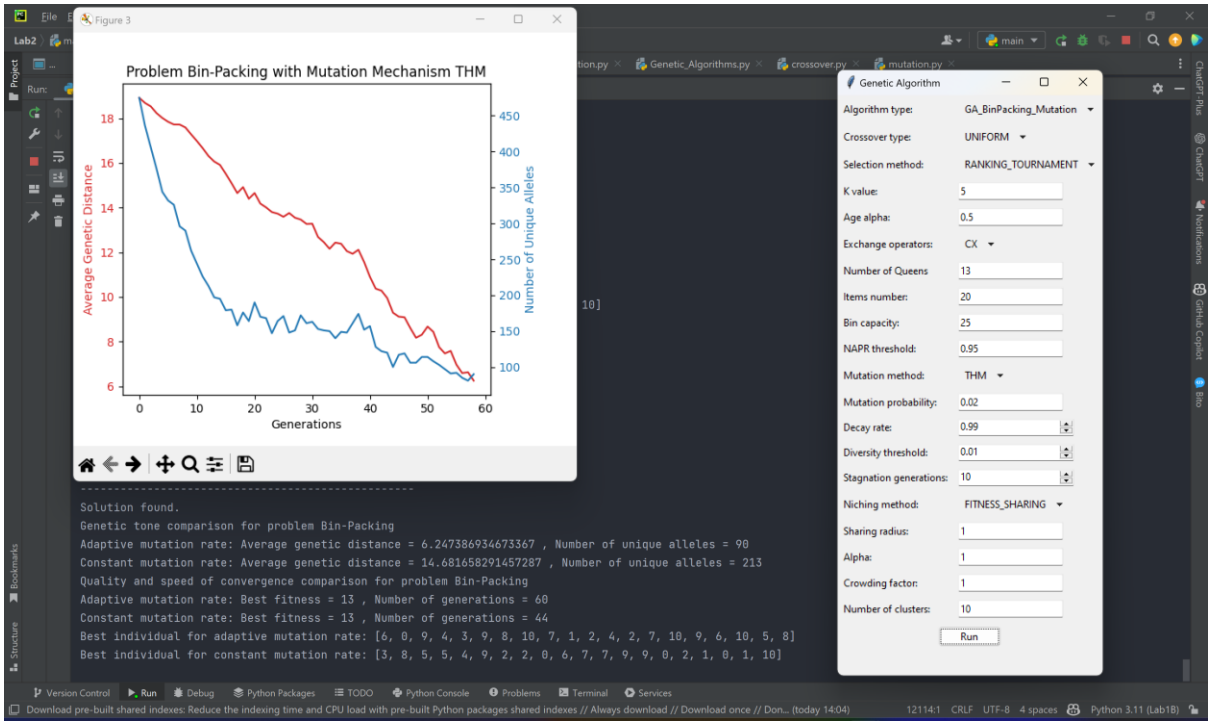


- Mutation probability: 0.5
- Threshold: 0.85



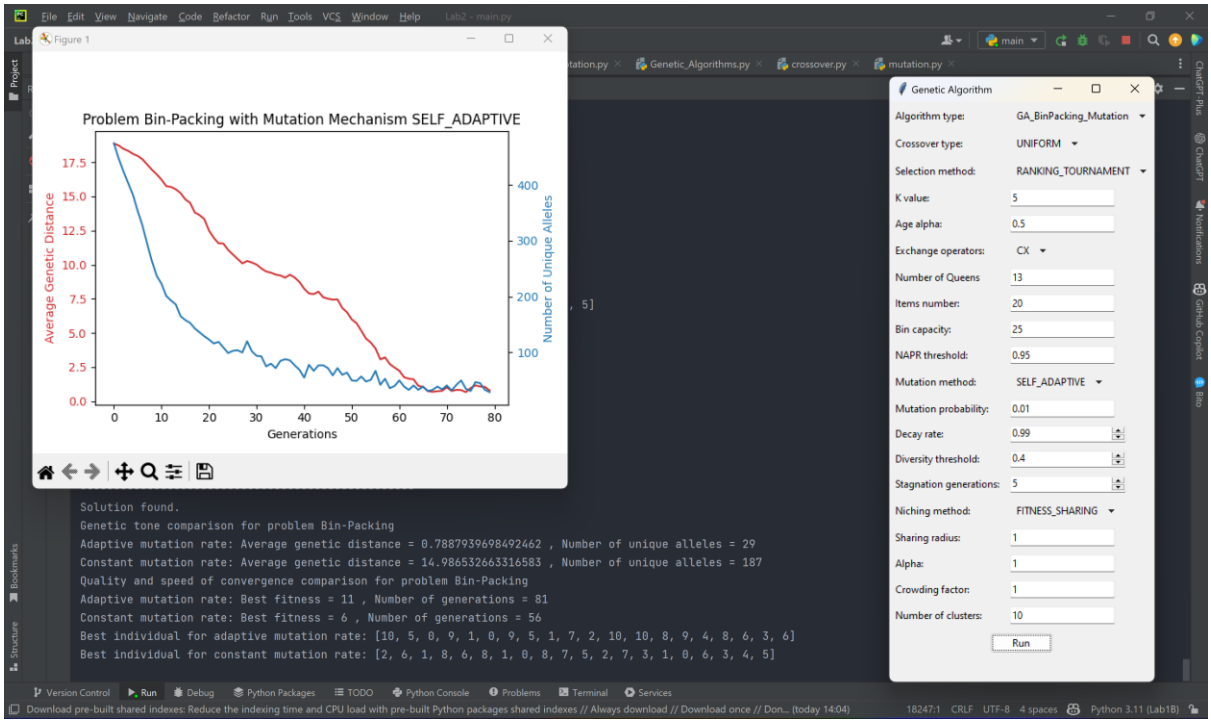
Mutation method: **"THM"**

- Mutation probability: 0.02
- Decay rate: 0.99
- Threshold: 0.01
- Stagnation threshold: 10



Mutation method: **"SELF_ADAPTIVE"**

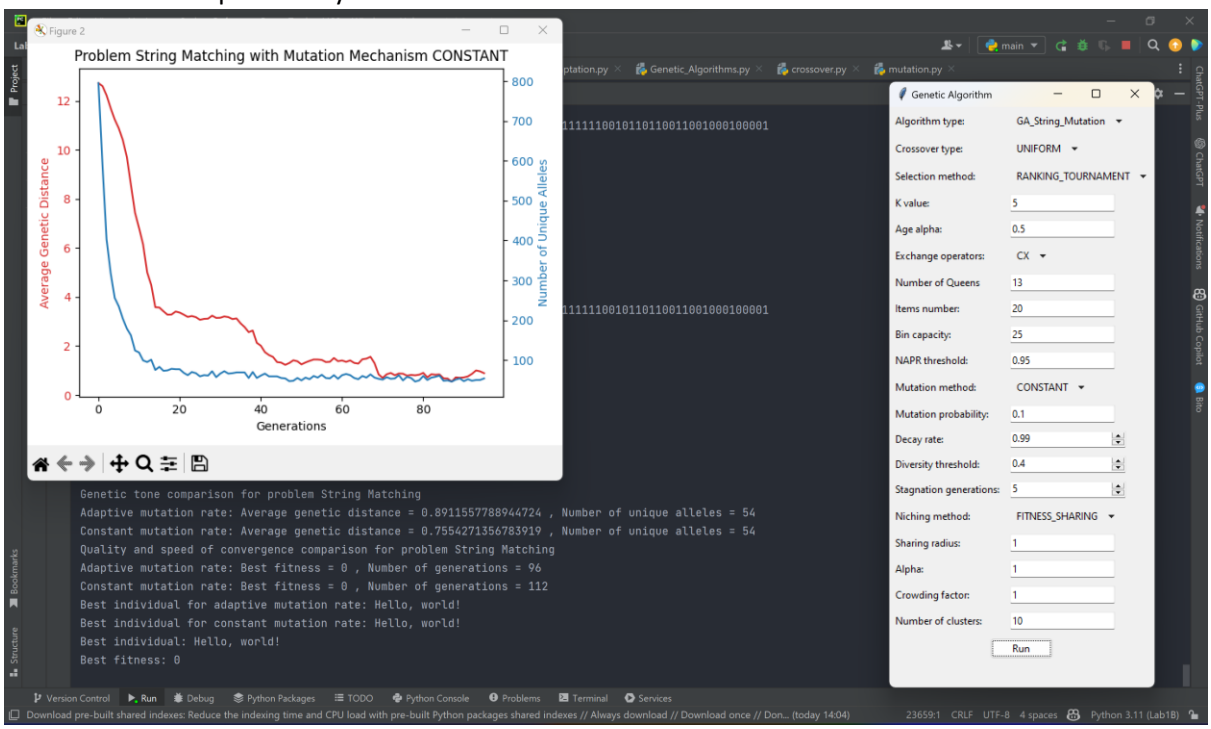
- Mutation probability: 0.01
- Decay rate: 0.99
- Threshold: 0.4
- Stagnation threshold: 5



String matching:

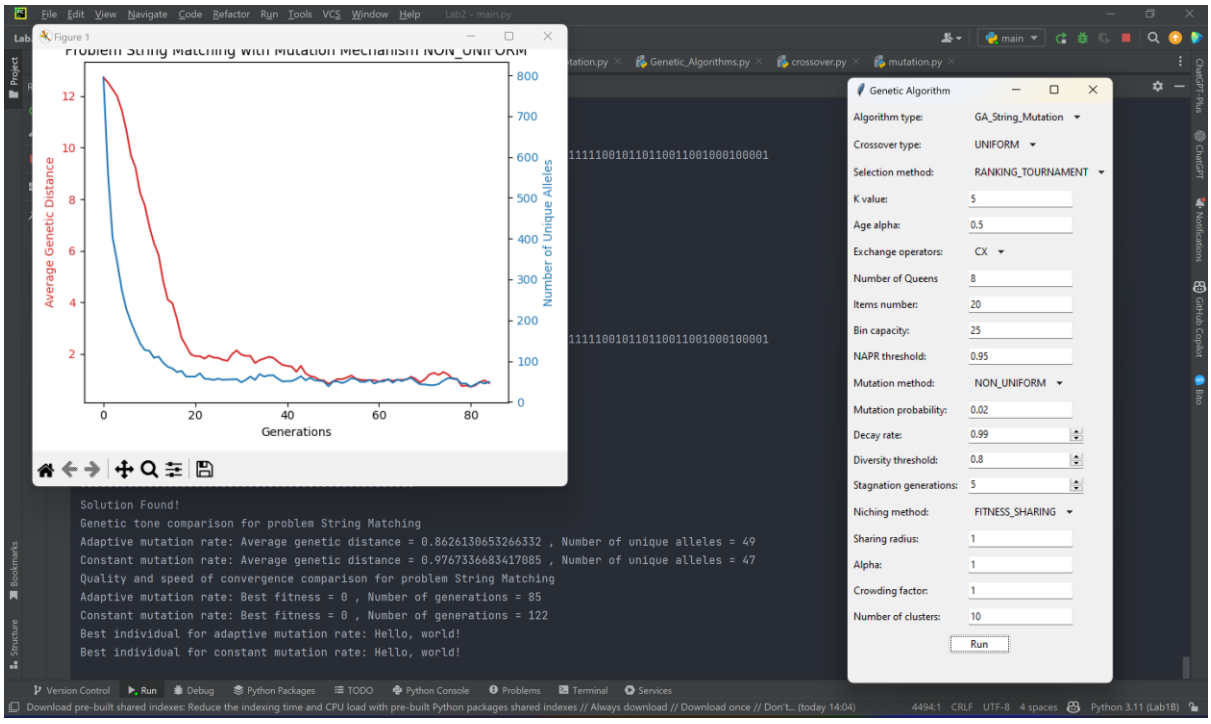
Mutation method: **"CONSTANT"**

- Mutation probability: 0.1



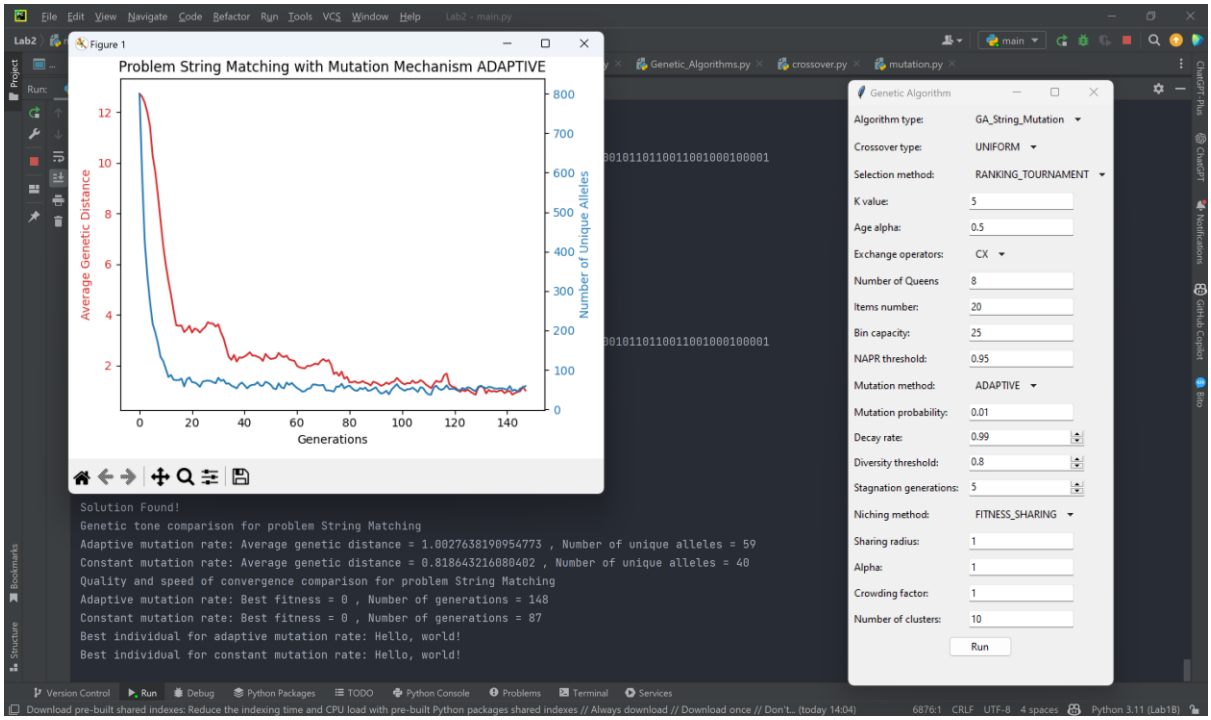
Mutation method: **"NON_UNIFORM"**

- Mutation probability: 0.02
- Decay rate: 0.99



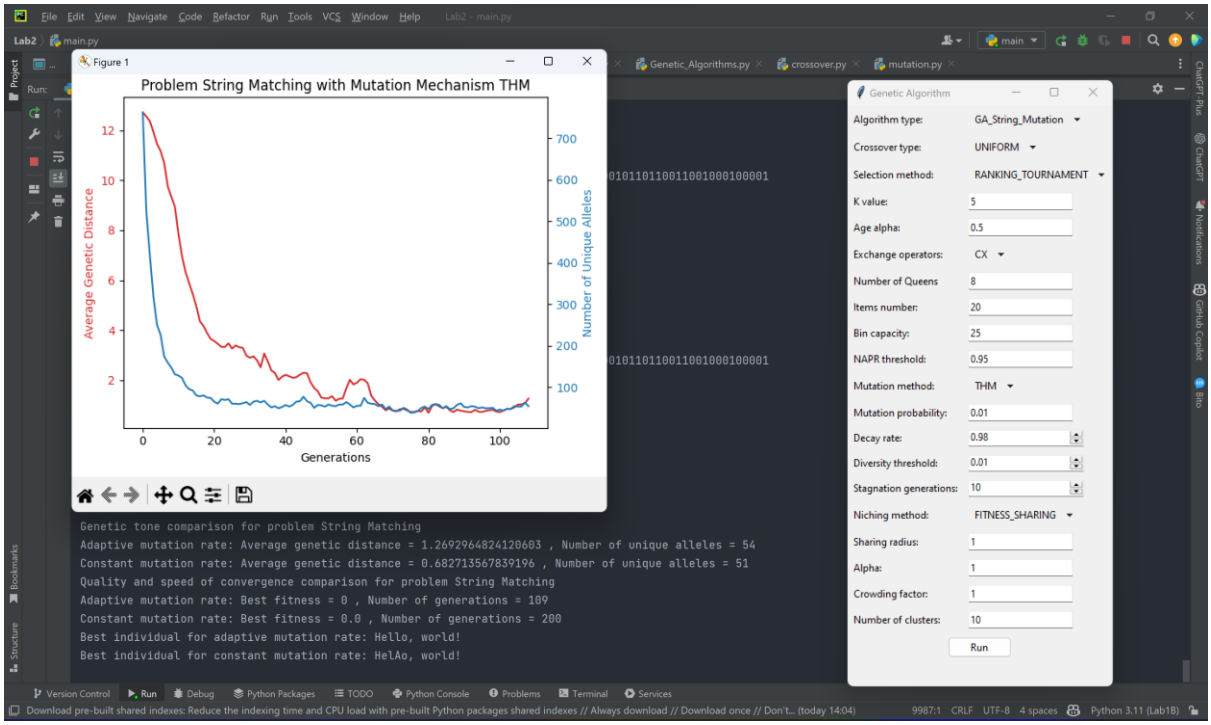
Mutation method: **"ADAPTIVE"**

- Mutation probability: 0.01
- Threshold: 0.8



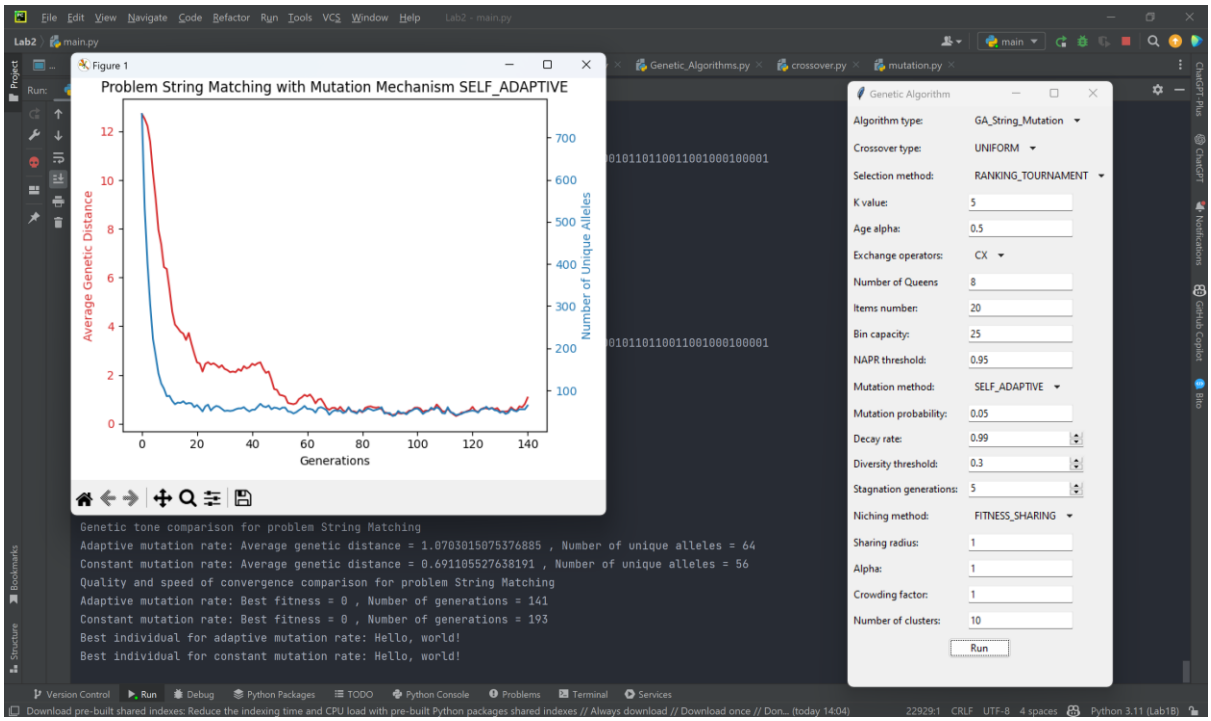
Mutation method: **"THM"**

- Mutation probability: 0.01
- Decay rate: 0.98
- Threshold: 0.01
- Stagnation threshold: 10



Mutation method: **"SELF_ADAPTIVE"**

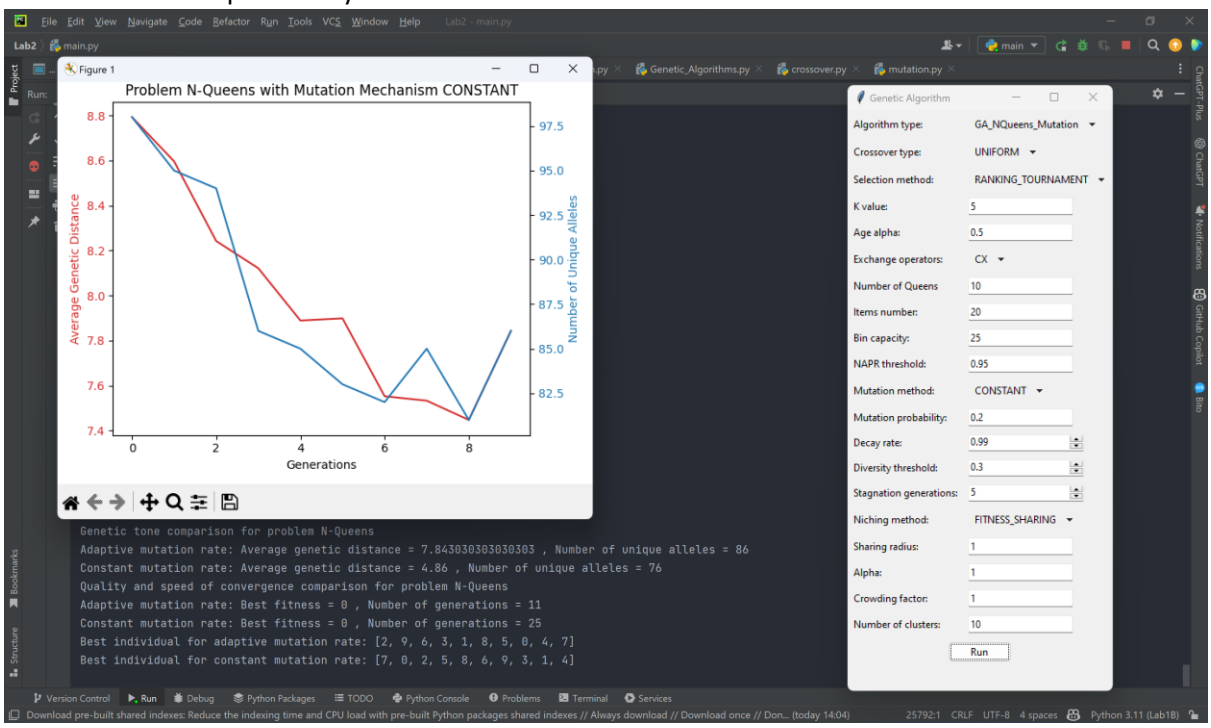
- Mutation probability: 0.05
- Decay rate: 0.99
- Threshold: 0.3
- Stagnation threshold: 5



N-Queens:

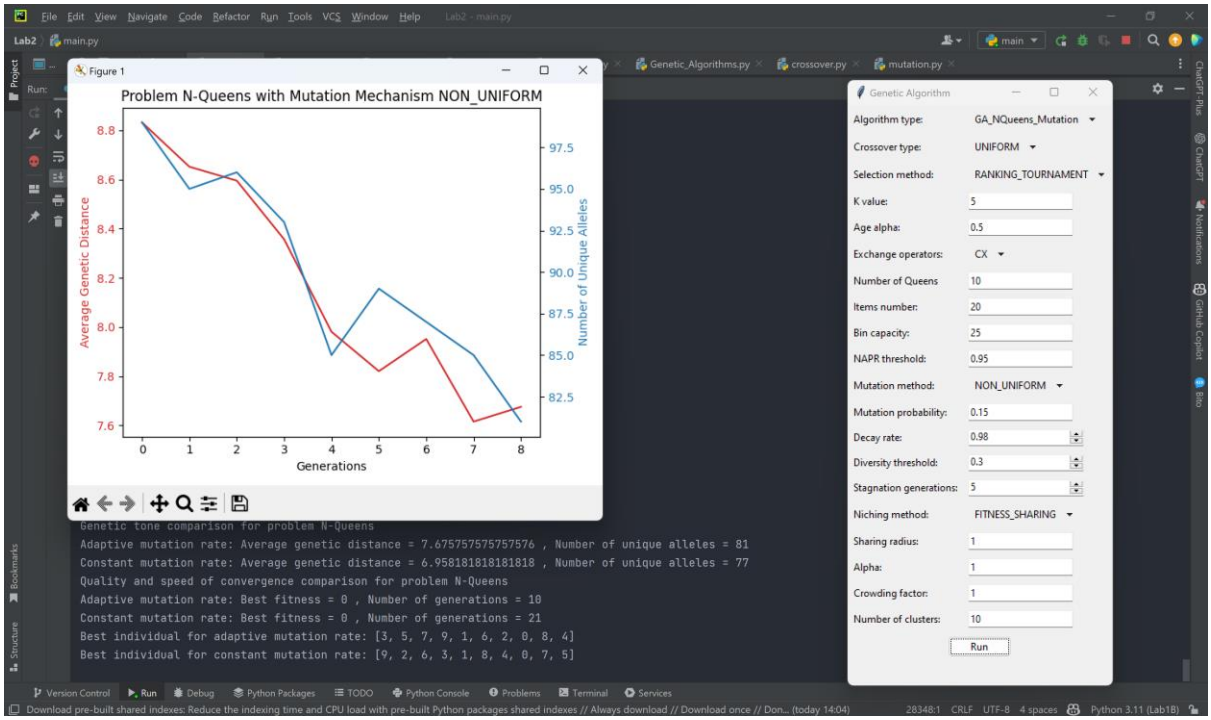
Mutation method: **" CONSTANT "**

- Mutation probability: 0.2



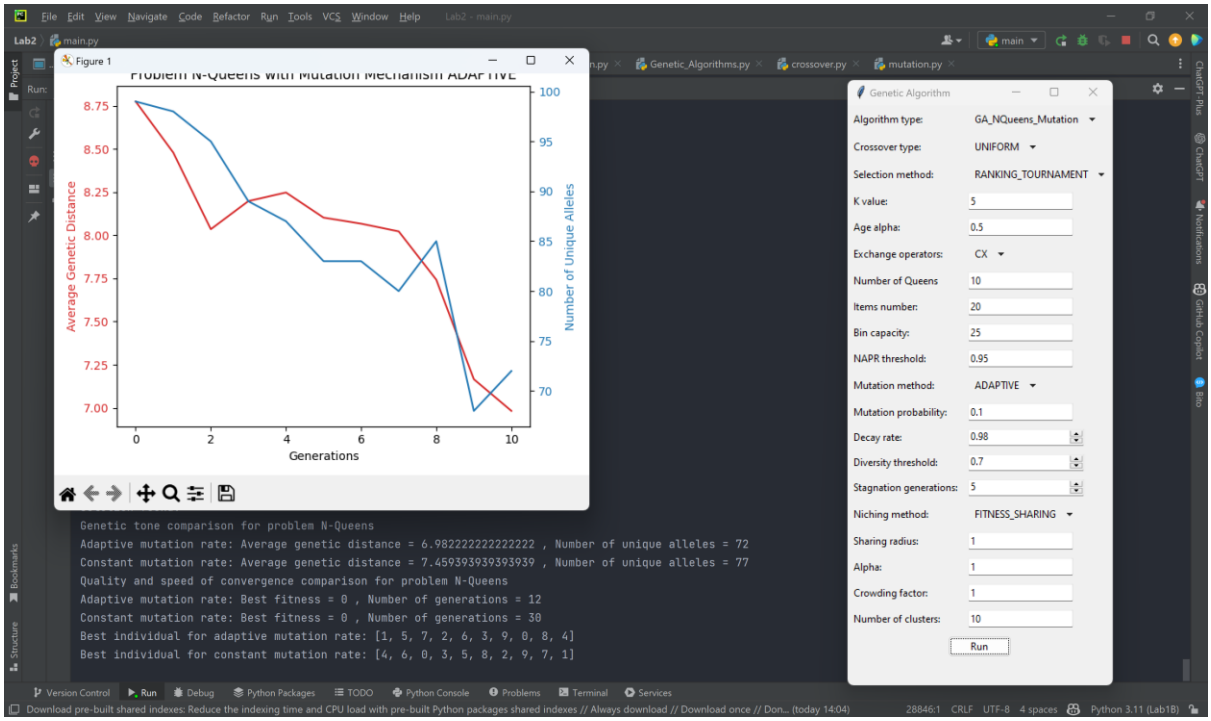
Mutation method: **"NON_UNIFORM"**

- Mutation probability: 0.15
- Decay rate: 0.98



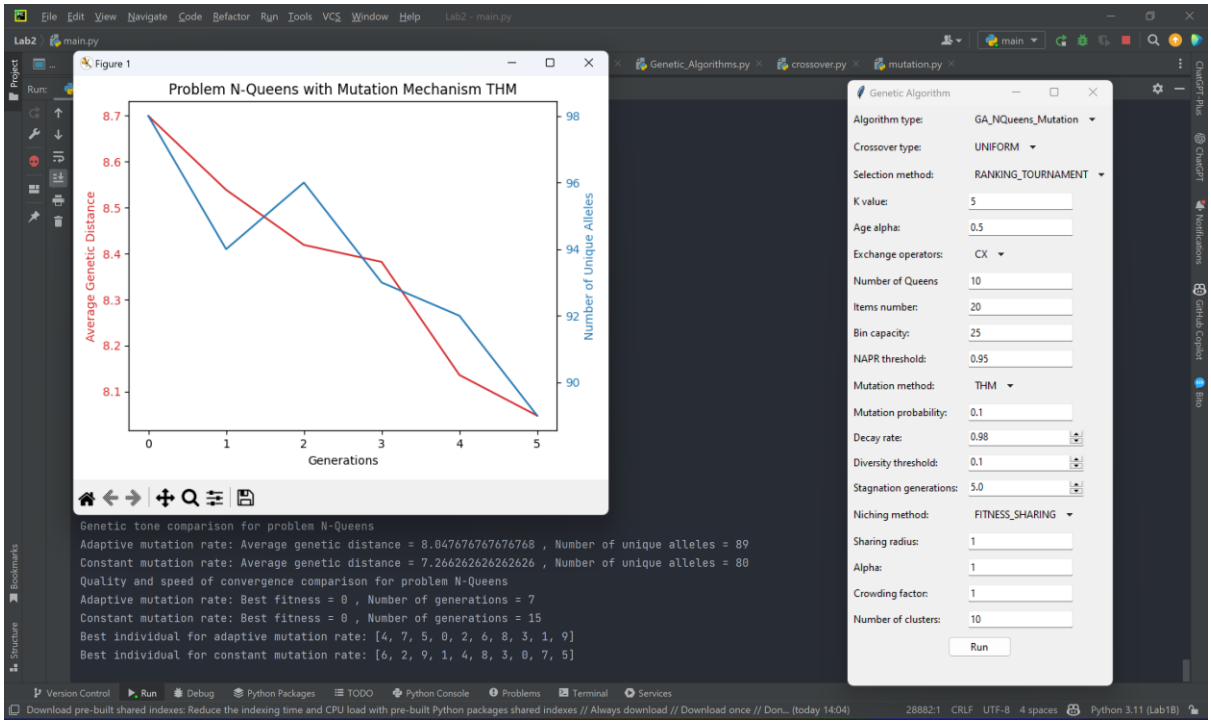
Mutation method: **"ADAPTIVE"**

- Mutation probability: 0.1
- Threshold: 0.7



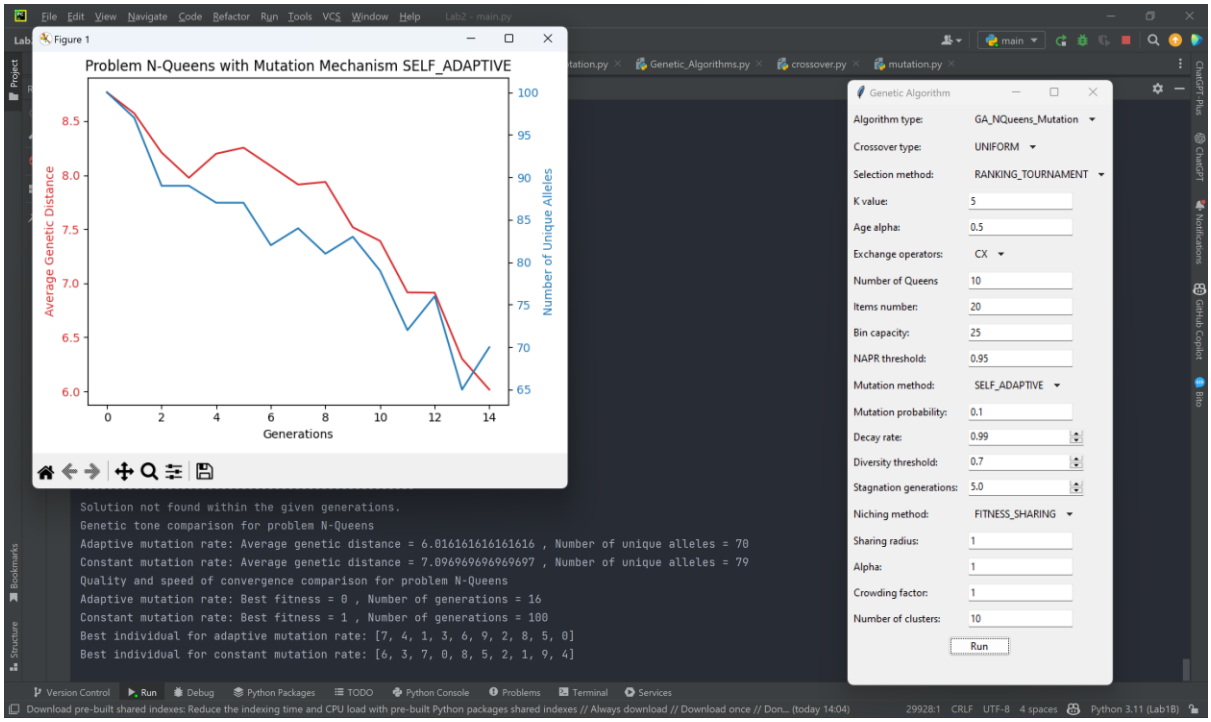
Mutation method: **"THM"**

- Mutation probability: 0.01
- Decay rate: 0.98
- Threshold: 0.1
- Stagnation threshold: 5



Mutation method: **"SELF_ADAPTIVE"**

- Mutation probability: 0.1
- Decay rate: 0.99
- Threshold: 0.7
- Stagnation threshold: 5



5. שלבו את אלגוריתמי המחיצות NIECHING, צפיפות CROWDING (**לא דטרמיניסטי**) ופיצול זנים SPECIATION עם CLUSTERING (שלושה אלגוריתמים סה"כ) במנוע שפיתחתם

For this section we added Niching.py file that includes these implementations:

1. **fitness_sharing**: This function calculates the shared fitness for each individual in the population. The shared fitness is calculated by dividing the fitness of each individual by the niche count, which represents the number of individuals in its neighbourhood. The niche count is computed using the sharing function that incorporates the distance between fitness values, the sharing radius (sigma_share), and the shape of the sharing function (alpha).

```python
def fitness_sharing(population, fitnesses, sigma_share, alpha):
    shared_fitnesses = []
    epsilon = 1e-6
    for i, fit_i in enumerate(fitnesses):
        niche_count = sum([sharing_function(fit_i, fit_j, sigma_share, alpha) for j, fit_j in enumerate(fitnesses) if i != j])
        shared_fitness = fit_i / (niche_count + epsilon)
        shared_fitnesses.append(shared_fitness)
    return shared_fitnesses


def sharing_function(fit_i, fit_j, sigma_share, alpha):
    d = abs(fit_i - fit_j)
    if d < sigma_share:
        return 1 - (d / sigma_share) ** alpha
    else:
        return 0
```

2. **crowding**: This function calculates the crowded fitness for each individual in the population. Crowding is used to maintain diversity in the population by considering the fitness of an individual relative to its neighbouring individuals. Crowding distances are calculated for each individual by summing the distances to the nearest crowding_factor number of neighbors in the population. The crowded fitness is then calculated by dividing the original fitness by (1 + crowding distance).

```python
def crowding(population, fitnesses, crowding_factor):
    population_size = len(population)
    crowding_distances = [0] * population_size

    for i in range(population_size):
        distances = [abs(fitnesses[i] - fitnesses[j]) for j in range(population_size) if i != j]
        distances.sort()
        crowding_distances[i] = sum(distances[:crowding_factor])

    return [fitness / (1 + crowding_distance) for fitness, crowding_distance in zip(fitnesses, crowding_distances)]
```

3. **speciation_with_clustering**: This function divides the population into species (clusters) based on their fitness values using the specified clustering algorithm (e.g., K-means). The fitness values are then shared among individuals within the same species. The shared fitness for each individual is calculated by dividing its original fitness by the size of its species. If the size of the species is 1, the shared fitness remains the same as the original fitness.

```
def speciation_with_clustering(population, fitnesses, num_species, clustering_algorithm='k-means'):
    if clustering_algorithm == 'k-means':
        kmeans = KMeans(n_clusters=num_species, random_state=0, n_init=10).fit(np.array(fitnesses).reshape(-1, 1))
        species_labels = kmeans.labels_
    else:
        raise ValueError(f"Unknown clustering algorithm: {clustering_algorithm}")

    species_fitnesses = [[] for _ in range(num_species)]

    for i, species_label in enumerate(species_labels):
        species_fitnesses[species_label].append(fitnesses[i])

    shared_fitnesses = []
    for i, species_label in enumerate(species_labels):
        species_fitness = fitnesses[i]
        species_size = len(species_fitnesses[species_label])

        if species_size > 1:
            shared_fitness = species_fitness / species_size
        else:
            shared_fitness = species_fitness

        shared_fitnesses.append(shared_fitness)

    return shared_fitnesses
```

6. השוו ביצועי אלגוריתמים אלה באמצעות המנוע על בעיית ה BIN PACKING מבחינת **איכות** אוסף הפתרונות השונים המיטביים המתקבל בסוף האבולוציה **ומה מידת הדמיון** בין פתרונות אלה. ציינו מהם הפרמטרים שבחרתם עבור כל אלגוריתם וכן פרטים ספציפיים לכל אלגוריתם כגון מספר המחיצות, כמות הCLUSTERS האופטימלית וכו'

Here's an explanation of each parameter and some example values:

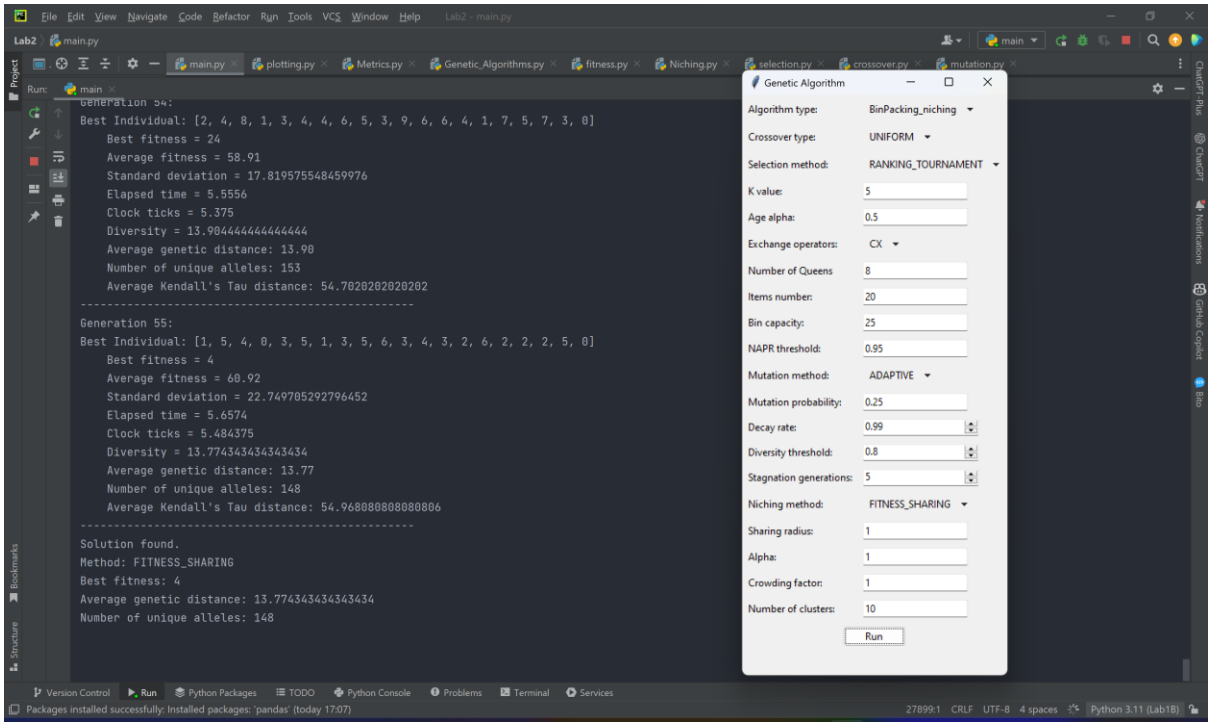
1. sharing_radius: This parameter is used in the **fitness sharing niching method**. It determines the radius around an individual within which other individuals are considered to belong to the same niche. A larger sharing radius means that more individuals will be considered part of the same niche and will have their fitness adjusted accordingly. A smaller sharing radius will result in fewer individuals being considered part of the same niche, allowing for more diversity in the population.
Example values: 0.1, 0.5, 1, 2.

2. alpha: This parameter is also used in the **fitness sharing niching method**. It represents the sharing function's shape, which determines how the fitness is adjusted for individuals within a niche. A larger alpha value means that the fitness adjustment will be more severe, potentially leading to more diverse solutions. A smaller alpha value will result in a more gradual fitness adjustment, allowing for more similar solutions within a niche.
Example values: 0.5, 1, 1.5, 2.

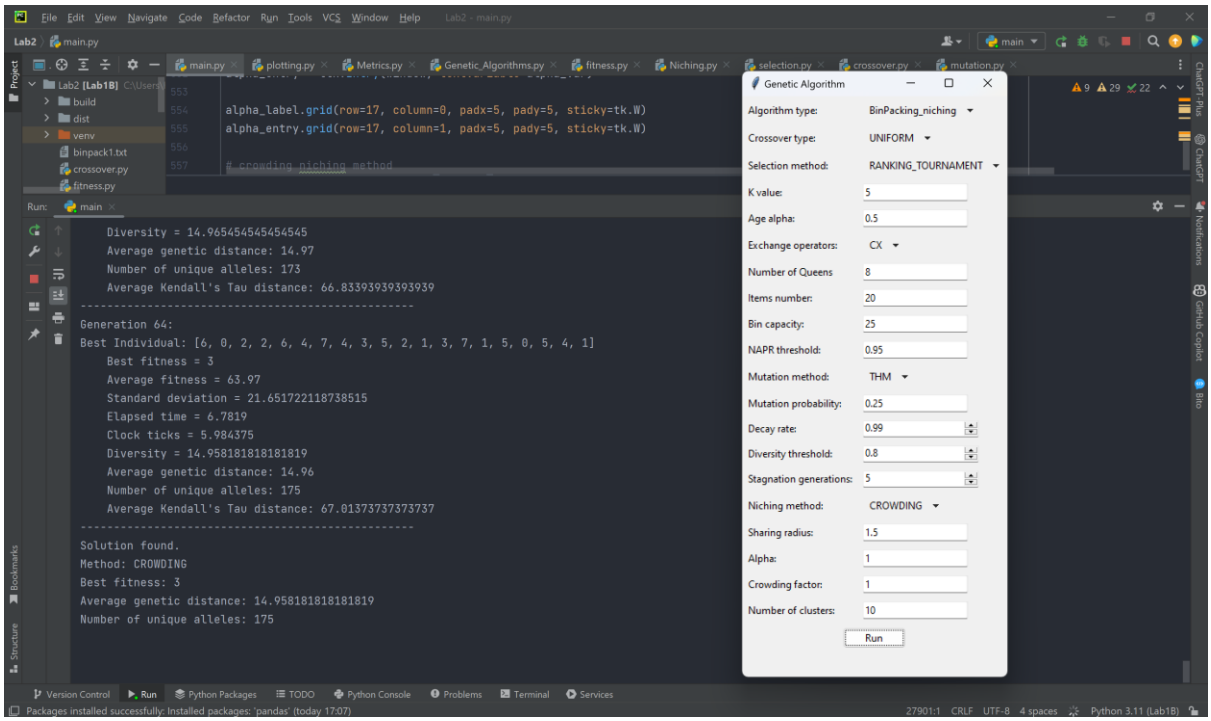
3. crowding_factor: This parameter is used in the **crowding niching method**. It controls the strength of the crowding effect on an individual's fitness. A larger crowding factor will result in a more significant fitness adjustment for crowded individuals, promoting more diversity in the population. A smaller crowding factor will have a more subtle effect on an individual's fitness, allowing for more similar solutions to coexist.
Example values: 0.1, 0.5, 1, 2.

4. num_species: This parameter is used in the speciation with **clustering niching method**. It defines the number of species (or clusters) that the population will be divided into. A larger number of species will result in more distinct groups of solutions, potentially leading to a more diverse set of solutions. A smaller number of species may cause more similar solutions to be grouped together, reducing the overall diversity.
Example values: 5, 10, 15, 20.

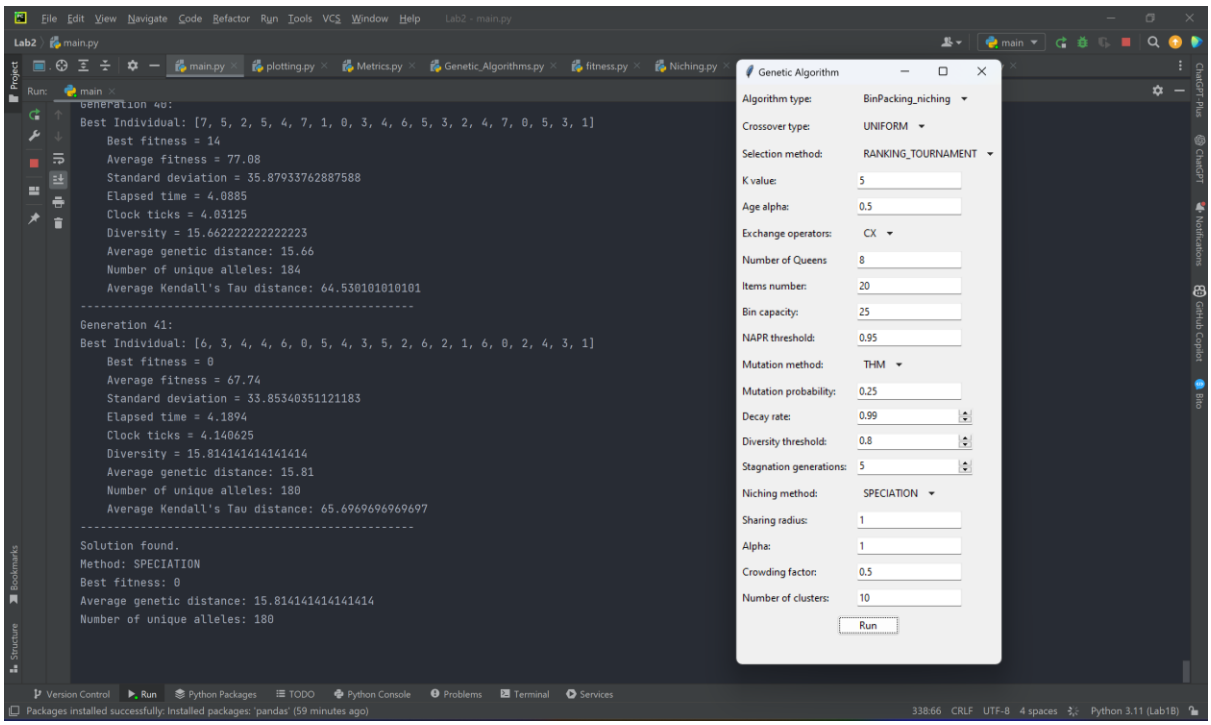
Fitness sharing:



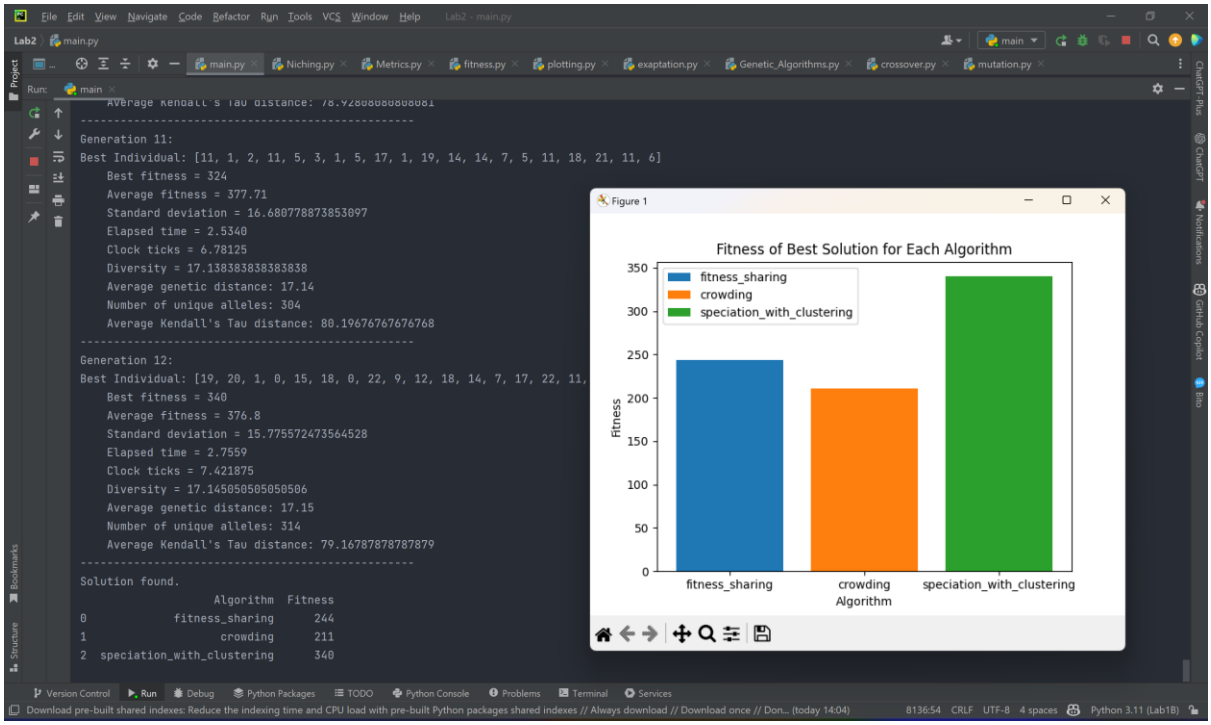
Crowding:



Speciation:



I also made a comparison between the three, and we got that the "speciation_with_clustering" algorithm is the most effective of the three algorithms we have tested for the bin packing problem, followed by the "fitness_sharing" algorithm, and the "crowding" algorithm.



7. ממשו את אלגוריתם האי EXAPTATION בהתבסס על המנוע הגנטי שלכם עבור האופטימיזציה של שתי הפונקציות הבאות

$f(x, y) = x^2 + y^2$ מעגל עם אילוץ ש R<=3

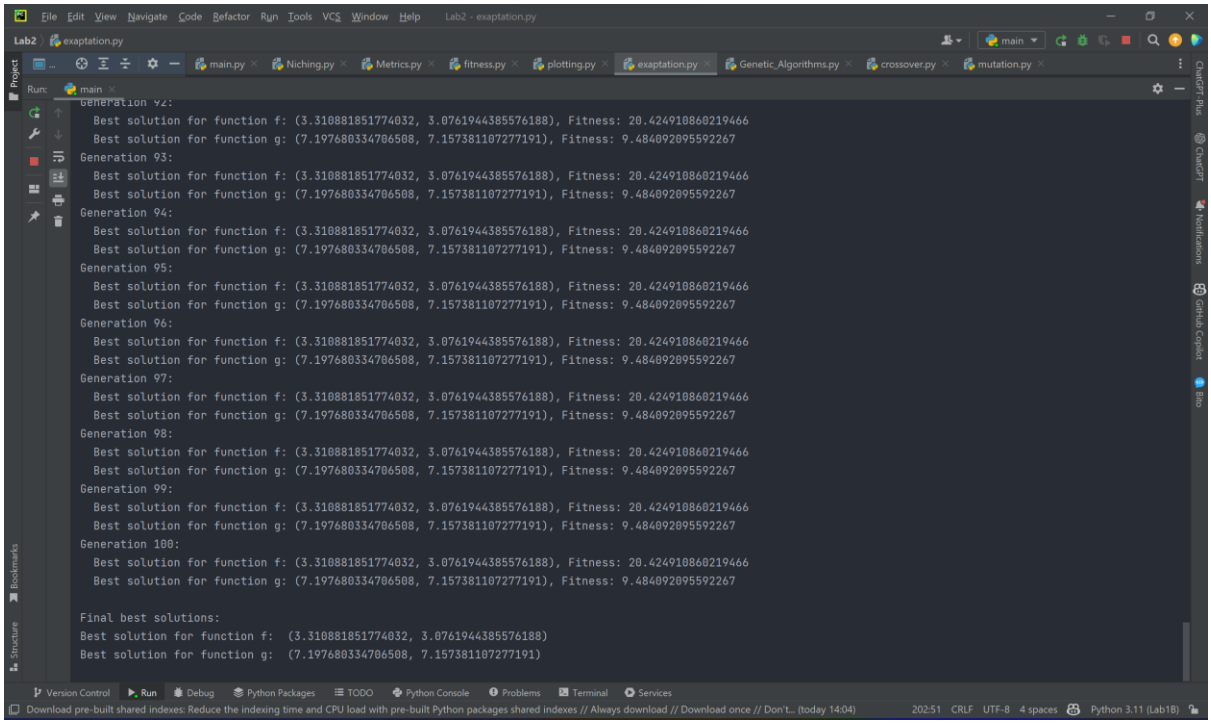
$g(x, y) = (x - 5)^2 + (y - 5)^2$ מעגל עם אילוץ ש R<=2

- עליכם לממש שני איים שונים
- כל פונקציה תאופטם באי נפרד
- לקבע את ההיפר-פרמטרים של כל אי
- גנים יעברו בהתאם לVIABILITY שלהם
- מהגרים חזקים יחליפו פרטים חלשים

For this section we added **exaptation.py** file that includes the implementations of the EXAPTATION algorithm:

We implemented the EXAPTATION algorithm using a genetic engine to optimize the two given functions, f(x, y) and g(x, y), with specific constraints. The algorithm used two separate islands to optimize each function independently. We designed the algorithm to perform selection, crossover, and mutation within each island, and also included a migration step to transfer viable individuals between islands based on their viability. We tried to improve it to the best way we could to reach the target.

These are the results we got:



8. סכמו את המסקנות שלכם מהתהליך האבולוציוני שהרצתם ע"ס הסעיף הקודם

Conclusions from the evolutionary process:

1. The two-island approach allows each function to be optimized independently while still enabling the exchange of potentially beneficial individuals between islands.
2. The migration step ensures that useful genetic information can be shared between the two islands, potentially accelerating the optimization process.
3. The viability function acts as a gatekeeper, ensuring that only individuals that meet the specific constraints of each island can migrate, preventing the dilution of good solutions.

In summary, the island model genetic algorithm with tournament selection, single-point crossover, and mutation, along with elitism and periodic migration, effectively optimized the two functions under their respective constraints. This approach allowed for greater exploration and diversity in the populations, helping to avoid premature convergence and find good solutions to the optimization problem.