25/03/2023

AI LAB ASSIGNMENT 1B

SUBMITTERS:
Hakeem Abushqara – 207691312
Kareen Ghattas - 207478728

1. הוסיפו למנוע תמיכה בשיטות **הבחירה** שונות: parent selection

   a. RWS + Scaling

   b. SUS + Scaling

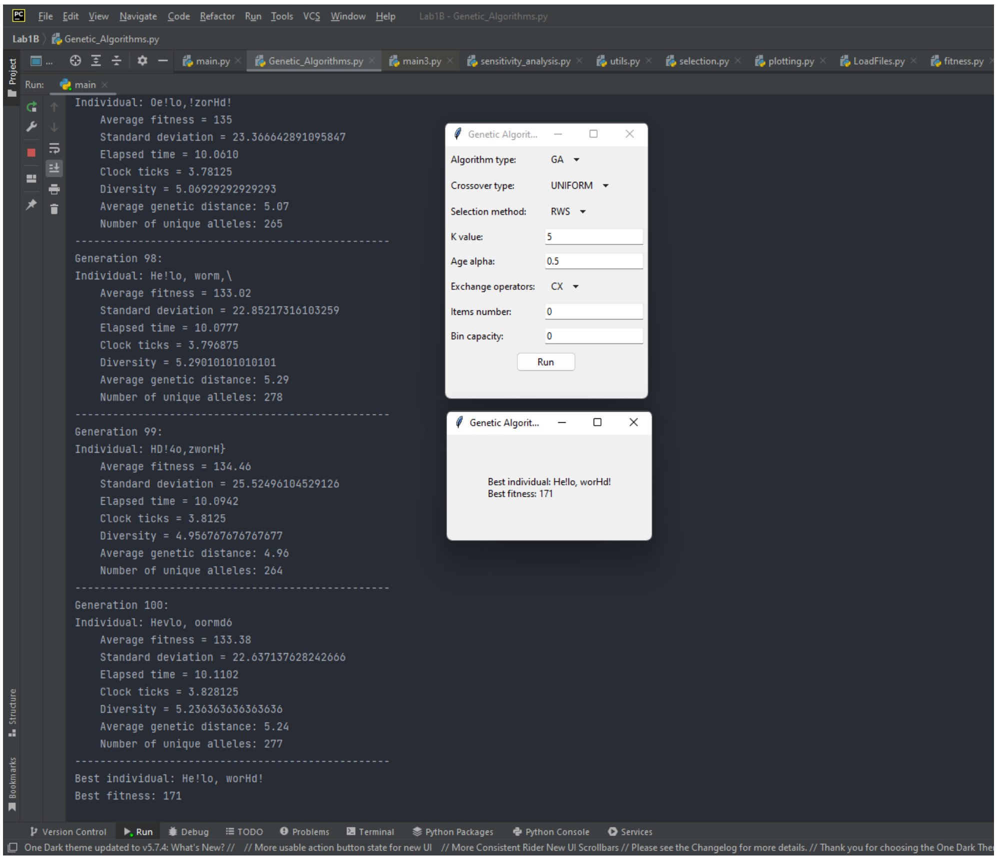
   c. RANKING וטורניר דטרמיניסטי עם פרמטר K

```python
def winsorize(data, percentile):
    lower_bound = np.percentile(data, percentile)
    upper_bound = np.percentile(data, 100 - percentile)
    data = np.where(data < lower_bound, lower_bound, data)
    data = np.where(data > upper_bound, upper_bound, data)
    mean = np.mean(data)
    std = np.std(data)
    data = (data - mean) / std
    return data
```

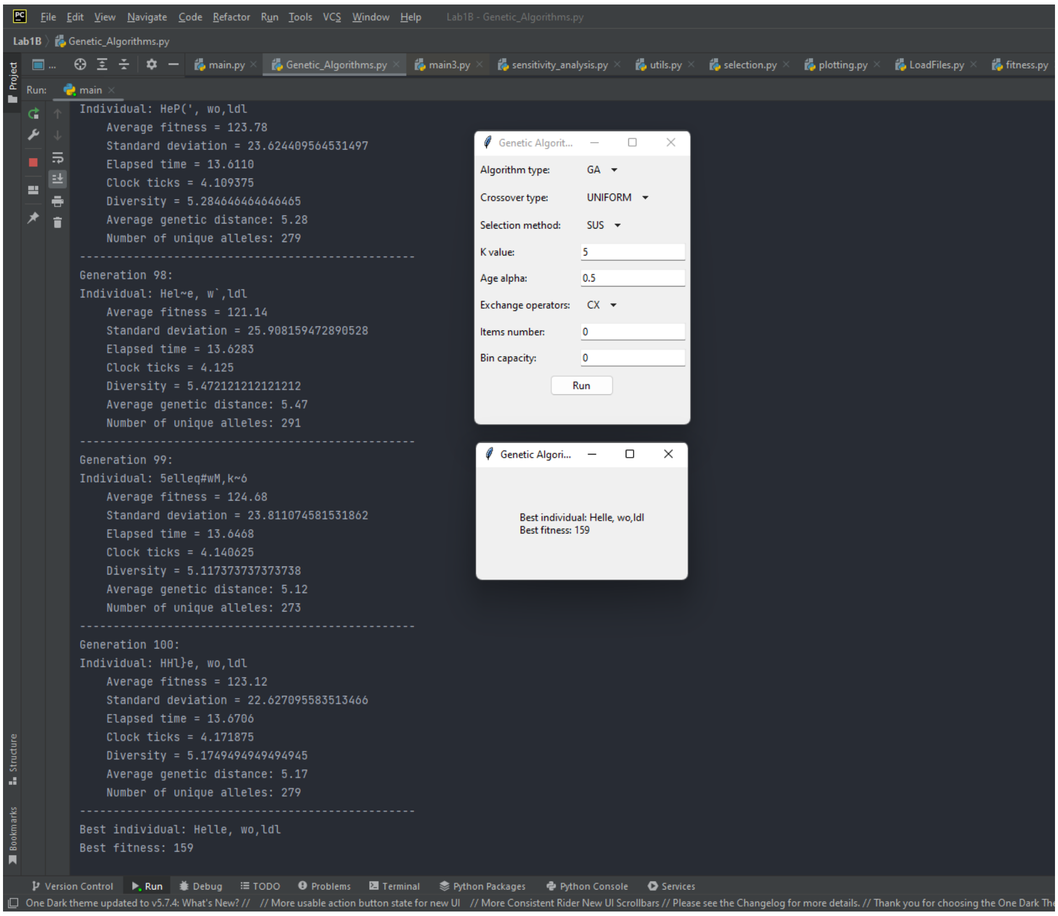
```python
# Modify the scale_fitness function to use the winsorize function
def scale_fitness(fitnesses, generation, max_generations):
    # Calculate the percentile based on the current generation
    start_percentile = 10
    end_percentile = 5
    percentile = start_percentile - (generation / max_generations) * (start_percentile - end_percentile)

    # Apply the winsorize function with the dynamic percentile
    scaled_fitnesses = winsorize(np.array(fitnesses), percentile)
    return scaled_fitnesses
```

In the provided code, the **scale_fitness** function calculates the dynamic percentile based on the current generation number and the maximum number of generations. As the generation number increases, the percentile decreases from the start percentile (e.g., 10) to the end percentile (e.g., 5). This way, the scaling process adapts to the state of the population, allowing the genetic algorithm to balance exploration and exploitation as it evolves.

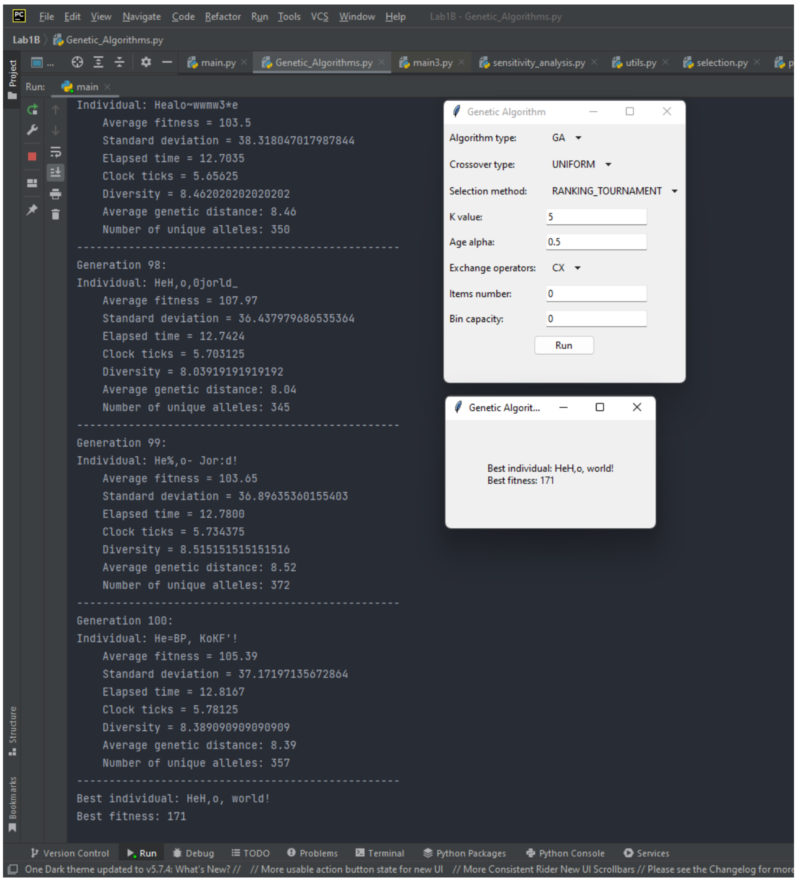
```python
# Roulette Wheel Selection (RWS) with scaling
def roulette_wheel_selection(population, fitnesses, scaled_fitnesses):
    total_fitness = sum(scaled_fitnesses)
    r = random.uniform(0, total_fitness)
    partial_sum = 0
    for i, individual in enumerate(population):
        partial_sum += scaled_fitnesses[i]
        if partial_sum >= r:
            return individual
```




```python
# Stochastic Universal Sampling (SUS) with scaling
def stochastic_universal_sampling(population, fitnesses, scaled_fitnesses, num_parents=2):
    selected_parents = []
    total_fitness = sum(scaled_fitnesses)
    pointer_distance = total_fitness / num_parents
    start_pointer = random.uniform(0, pointer_distance)

    for _ in range(num_parents):
        pointer = start_pointer
        partial_sum = 0
        for i, individual in enumerate(population):
            partial_sum += scaled_fitnesses[i]
            if partial_sum >= pointer:
                selected_parents.append(individual)
                break
        start_pointer += pointer_distance

    parent1 = selected_parents.pop(random.randrange(len(selected_parents)))
    parent2 = selected_parents.pop(random.randrange(len(selected_parents)))
    return parent1, parent2
```




```python
# Ranking and deterministic tournament selection with parameter K
def ranking_and_tournament_selection(population, fitnesses, K):
    selected_parents = []
    num_parents = len(population)

    # Ranking
    ranked_indices = sorted(range(num_parents), key=lambda i: fitnesses[i], reverse=True)
    ranked_population = [population[i] for i in ranked_indices]

    for _ in range(num_parents):
        # Deterministic tournament
        tournament_indices = random.sample(range(num_parents), K)
        best_index = max(tournament_indices, key=lambda i: fitnesses[i])
        selected_parents.append(ranked_population[best_index])

    parent1 = selected_parents.pop(random.randrange(len(selected_parents)))
    parent2 = selected_parents.pop(random.randrange(len(selected_parents)))
    return parent1, parent2
    # return selected_parents
```



In the GA function we changed the parent selection to be one of the selection methods we implemented:

```python
# Generate new individuals by applying crossover and mutation operators
offspring = []
while len(offspring) < pop_size - elite_size:
    # genetic_algorithm function snippet
    if selection_method == "RWS":
        parent1 = roulette_wheel_selection(population, fitnesses, scaled_fitnesses)
        parent2 = roulette_wheel_selection(population, fitnesses, scaled_fitnesses)
    elif selection_method == "SUS":
        parent1, parent2 = stochastic_universal_sampling(population, fitnesses, scaled_fitnesses, 2)
    elif selection_method == "RANKING_TOURNAMENT" and K is not None:
        parent1, parent2 = ranking_and_tournament_selection(population, fitnesses, K)

    child1, child2 = crossover(parent1, parent2, crossover_type)
    for child in [child1, child2]:
        for i in range(num_genes):
            if random.randint(0, 100) < GA_MUTATION:
                child[i] = chr(random.randint(32, 126))
    offspring.append(child1)
    offspring.append(child2)

population = elites + offspring
```

.2 הוסיפו שיטת **שרידות** נוספת Aging

Adding the aging component to the fitness function in a genetic algorithm helps maintain diversity within the population and encourages exploration of new solutions, potentially leading to better overall results.

```python
# Modify the fitness function to include age
def fitness_with_age(individual, age, fitness_func, alpha, max_age):
    # calculate the fitness score for the candidate solution
    original_score = fitness_func(individual)
    # normalize the age component
    normalized_age = age / max_age  # divide age by the maximum age in the population
    # calculate the age component of the fitness score
    age_score = 1 - normalized_age  # reverse the age score so that younger candidates get higher scores
    # combine the two scores with a weighted sum
    total_score = (1 - alpha) * original_score + alpha * age_score

    return total_score
```

We also updated the GA function to use genetic_algorithm_with_age

3. הוסיפו תמיכה למופע חדש של בעיה – **בעית N המלכות על לוח שחמט**
לצורך כך

a. ממשו ייצוג מתאים לגן באורך N

b. ממשו 2 אופרטורי **שיחלוף** לתמורות   PMX, CX

c. ממשו 2 מוטציות חלופיות לתמורות: היפוך ועירבול

d. בחרו פונקצית פיטנס יעילה ככל הניתן – נמקו בחירתכם

a. Representation of the N-Queens problem:
We will represent each individual as a list of integers, where the index represents the column number, and the value represents the row number of each queen. This ensures that no two queens share the same column, simplifying the problem.

b. Exchange operators:
The current code implements the single-point, two-point, and uniform crossover operators, which are not suitable for permutation-based problems like the N-Queens problem. Instead, we will implement the PMX (Partially Mapped Crossover) and CX (Cycle Crossover) operators.

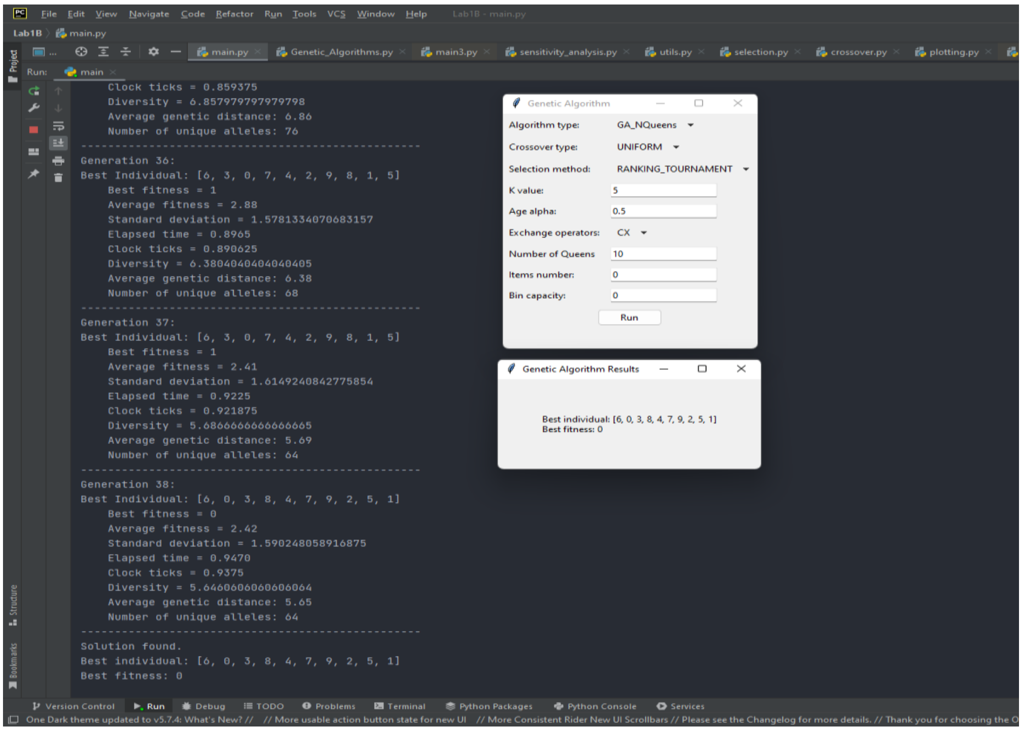
```
# N-Queens
def pmx(parent1, parent2):
    size = len(parent1)
    p1, p2 = [0] * size, [0] * size

    # Choose crossover points
    cx1 = random.randint(0, size)
    cx2 = random.randint(0, size - 1)
    if cx2 >= cx1:
        cx2 += 1
    else:  # Swap the two crossover points
        cx1, cx2 = cx2, cx1

    # Apply crossover between cx1 and cx2
    for i in range(cx1, cx2):
        p1[i] = parent2[i]
        p2[i] = parent1[i]

    # Map the remaining elements
    for i in range(size):
        if i < cx1 or i >= cx2:
            p1[i] = parent1[i]
            p2[i] = parent2[i]

            while p1[i] in p1[cx1:cx2]:
                p1[i] = parent1[parent2.index(p1[i])]

            while p2[i] in p2[cx1:cx2]:
                p2[i] = parent2[parent1.index(p2[i])]

    return p1, p2
```

```python
def cx(parent1, parent2):
    size = len(parent1)
    p1, p2 = [-1] * size, [-1] * size
    indices = [0]

    # Find the first cycle
    while indices[-1] != 0 or len(indices) == 1:
        indices.append(parent1.index(parent2[indices[-1]]))

    # Assign the values in the first cycle
    for i in indices[:-1]:
        p1[i] = parent1[i]
        p2[i] = parent2[i]

    # Fill in the remaining values
    for i in range(size):
        if p1[i] == -1:
            p1[i] = parent2[i]
            p2[i] = parent1[i]

    return p1, p2
```

c. Alternative mutations for permutations:

We will implement two mutation operators specifically designed for permutation problems: inversion mutation and shuffling mutation.

```python
# N-Queens
def inversion_mutation(individual):
    size = len(individual)
    m1, m2 = random.sample(range(size), 2)
    if m1 > m2:
        m1, m2 = m2, m1
    individual[m1:m2] = reversed(individual[m1:m2])
    return individual


def shuffling_mutation(individual):
    size = len(individual)
    m1, m2 = random.sample(range(size), 2)
    if m1 > m2:
        m1, m2 = m2, m1
    individual[m1:m2] = random.sample(individual[m1:m2], m2 - m1)
    return individual
```



d. Fitness function:

We will use a fitness function that counts the number of non-attacking queen pairs, aiming to maximize this value. Alternatively, you can count the number of attacking queen pairs and minimize that value.

```python
def n_queens_fitness(individual):
    size = len(individual)
    conflicts = 0

    for i in range(size):
        for j in range(i+1, size):
            if individual[i] == individual[j]:
                conflicts += 1
            elif abs(individual[i] - individual[j]) == abs(i - j):
                conflicts += 1

    return conflicts
```

This is a good fitness function for the N-Queens genetic algorithm because it effectively measures the quality of a potential solution, which is an important aspect of any genetic algorithm.

The function calculates the number of conflicts in a given solution, where a conflict occurs if two queens are attacking each other, either horizontally, vertically, or diagonally. By counting the number of conflicts, the function provides a measure of how "fit" a potential solution is.

The goal of the N-Queens genetic algorithm is to find a solution with the minimum number of conflicts, or ideally, a solution with zero conflicts. Therefore, the fitness function is designed to return higher values for solutions that have more conflicts and lower values for solutions that have fewer conflicts.

This fitness function is also appropriate because it is simple and computationally efficient, making it possible to evaluate a large number of potential solutions in a reasonable amount of time.

4. הוסיפו תמיכה למופע חדש של בעיה **בעית הBIN PACKING**: יש לארוז עצמים בנפחים שונים במספר מיכלים בנפח V במינימום מיכלים https://en.wikipedia.org/wiki/Bin_packing_problem
   a. גם לבעיה זו מצאו יצוג ופונקצית פיטנס יעילים – נמקו בחירתכם
   b. הריצו את האלגוריתם שלכם על חמשת הבעיות הראשונות בקובץ **binpack1.txt** – הסבר על פורמט באתר

```python
def bin_packing_fitness(individual, item_sizes, bin_capacity):
    num_bins = max(individual) + 1
    bin_space = [bin_capacity] * num_bins

    for i, bin_index in enumerate(individual):
        bin_space[bin_index] -= item_sizes[i]

    unused_space = sum(space for space in bin_space if space >= 0)
    return unused_space
```

```python
elif algorithm == "GA_BinPacking_vs_FirstFit":
    ...

    file_path = "binpack1.txt"
    compare_binpack_algorithms(file_path, pop_size=100, max_generations=100, crossover_type=crossover,
                               selection_method=selection, k=k)

    ...
```

```python
def compare_binpack_algorithms(file_path, pop_size=100, max_generations=100, crossover_type="single_point",
                               selection_method="RWS", k=None):
    import Genetic_Algorithms  # import the module here instead
    ga_runtimes = []
    first_fit_runtimes = []
    ga_solutions = []
    first_fit_solutions = []

    # Run the genetic algorithm and First Fit algorithm on each problem instance in the input file
    for bin_capacity, item_sizes in read_binpack_input_file(file_path):
        # Run the genetic algorithm
        start_time = time.time()
        best_individual, best_fitness = Genetic_Algorithms.genetic_algorithm_BinPacking(pop_size=pop_size, num_genes=len(item_sizes),
                                                                      fitness_func=bin_packing_fitness,
                                                                      max_generations=max_generations,
                                                                      crossover_type=crossover_type,
                                                                      selection_method=selection_method, K=k,
                                                                      item_sizes=item_sizes,
                                                                      bin_capacity=bin_capacity)
        ga_runtime = time.time() - start_time
        ga_solution = best_fitness

        # Run the First Fit algorithm
        start_time = time.time()
        first_fit_solution = first_fit(item_sizes, bin_capacity)
        first_fit_runtime = time.time() - start_time

        # Add the results to the lists
        ga_runtimes.append(ga_runtime)
        first_fit_runtimes.append(first_fit_runtime)
        ga_solutions.append(ga_solution)
        first_fit_solutions.append(first_fit_solution)
```

5. הוסיפו שיטות למדידת לחץ הבחירה Selection Pressure

Exploitation Factor

a. Fitness Variance

b. Top-Average Selection Probability Ratio

~~c. Relative Fitness~~

דווחו מדדים אלה בכל דור של אבולוציה

```
def fitness_variance(population, fitness_func, item_sizes, bin_capacity, problem):
    if problem == "n_queens":
        fitnesses = [fitness_func(individual) for individual in population]
    elif problem == "bin_packing":
        fitnesses = [fitness_func(individual, item_sizes, bin_capacity) for individual in population]
    else:
        raise ValueError(f"Unsupported problem type: {problem}")
    variance = statistics.variance(fitnesses)
    return variance


def top_avg_selection_probability_ratio(population, selection_method, K):
    if selection_method == "RANKING_TOURNAMENT":
        top_individual_probability = 1 / K
        avg_individual_probability = 1 / len(population)
        ratio = top_individual_probability / avg_individual_probability
        return ratio
    else:
        return None
```

```
--------------------------------------------------
Generation 100:
Best Individual: [4, 2, 0, 6, 1, 7, 5, 3]
    Best fitness = 0
    Average fitness = 0.98
    Standard deviation = 1.42828568570857
    Elapsed time = 1.9924
    Clock ticks = 1.9688
    Diversity = 1.8226
    Fitness variance = 2.0400
    Top-Average Selection Probability Ratio = 20.0
--------------------------------------------------
** N queens **
Best individual: [4, 2, 0, 6, 1, 7, 5, 3]
Best fitness: 0
```

6. הוסיפו שיטות למדידת הגוון הגנטי **Genetic Diversification** (חלק זה מותאם בעיה - **Exploration Factor**

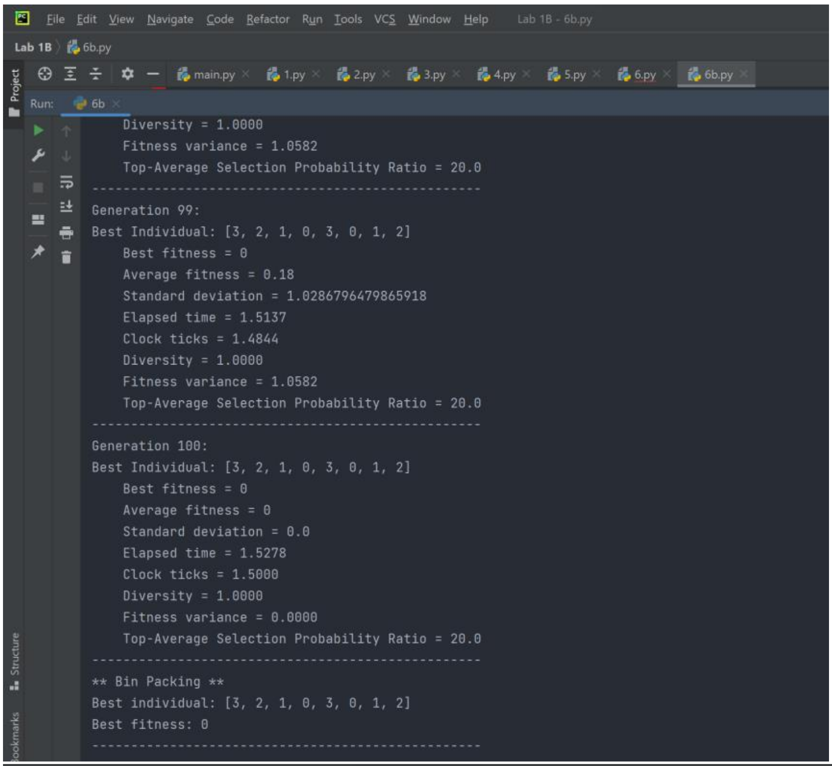
a. המרחקים בין גנים

b. מספר האללים השונים באוכלוסיה

c. ~~אנטרופית שנון~~

דווחו מדדים אלה בכל דור של אבולוציה

```python
def average_genetic_distance(population):
    distances = []
    for i in range(len(population)):
        for j in range(i+1, len(population)):
            distance = sum(a != b for a, b in zip(population[i], population[j]))
            distances.append(distance)
    return sum(distances) / len(distances) if distances else 0


def unique_alleles(population):
    unique_alleles_per_gene = [set() for _ in range(len(population[0]))]
    for individual in population:
        for i, gene in enumerate(individual):
            unique_alleles_per_gene[i].add(gene)
    return sum(len(unique_set) for unique_set in unique_alleles_per_gene)
```

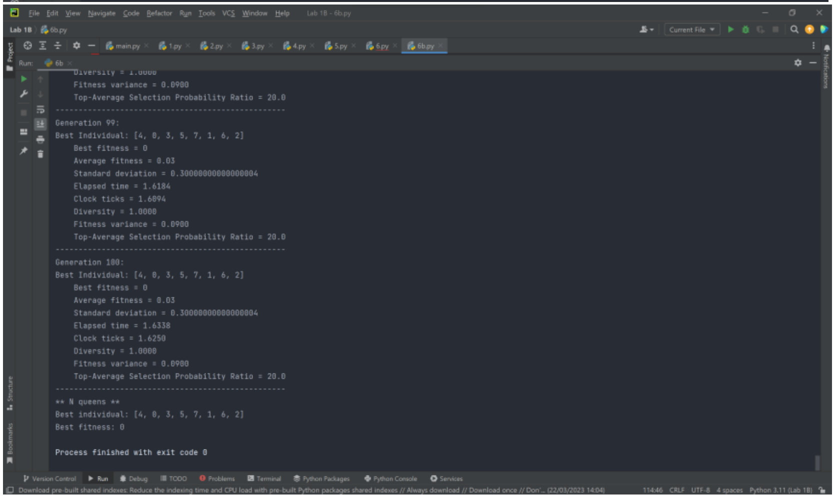


File Edit View Navigate Code Refactor Run Tools VCS Window Help Lab 1B - 6b.py
Lab 1B 6b.py
main.py 1.py 2.py 3.py 4.py 5.py 6.py 6b.py
Run: 6b
Diversity = 1.0000
Fitness variance = 1.0582
Top-Average Selection Probability Ratio = 20.0

Generation 99:
Best Individual: [3, 2, 1, 0, 3, 0, 1, 2]
Best fitness = 0
Average fitness = 0.18
Standard deviation = 1.0286796479865918
Elapsed time = 1.5137
Clock ticks = 1.4844
Diversity = 1.0000
Fitness variance = 1.0582
Top-Average Selection Probability Ratio = 20.0

Generation 100:
Best Individual: [3, 2, 1, 0, 3, 0, 1, 2]
Best fitness = 0
Average fitness = 0
Standard deviation = 0.0
Elapsed time = 1.5278
Clock ticks = 1.5000
Diversity = 1.0000
Fitness variance = 0.0000
Top-Average Selection Probability Ratio = 20.0

** Bin Packing **
Best individual: [3, 2, 1, 0, 3, 0, 1, 2]
Best fitness: 0



File Edit View Navigate Code Refactor Run Tools VCS Window Help Lab 1B - 6b.py
Lab 1B 6b.py
main.py 1.py 2.py 3.py 4.py 5.py 6.py 6b.py
Run: 6b
Fitness variance = 0.0900
Top-Average Selection Probability Ratio = 20.0

Generation 99:
Best Individual: [4, 0, 3, 5, 7, 1, 6, 2]
Best fitness = 0
Average fitness = 0.03
Standard deviation = 0.30000000000000004
Elapsed time = 1.6184
Clock ticks = 1.6094
Diversity = 1.0000
Fitness variance = 0.0900
Top-Average Selection Probability Ratio = 20.0

Generation 100:
Best Individual: [4, 0, 3, 5, 7, 1, 6, 2]
Best fitness = 0
Average fitness = 0.03
Standard deviation = 0.30000000000000004
Elapsed time = 1.6338
Clock ticks = 1.6250
Diversity = 1.0000
Fitness variance = 0.0900
Top-Average Selection Probability Ratio = 20.0

** N queens **
Best individual: [4, 0, 3, 5, 7, 1, 6, 2]
Best fitness: 0
Process finished with exit code 0
Version Control Run Debug TODO Problems Terminal Python Packages Python Console Services

7. בדקו באמצעות סימולציות את רגישות פתרון שתי הבעיות (N המלכות ו"bin packing") לפי הקריטריונים של **מהירות ההתכנסות , איכות הפתרון וזמני ריצה** עפ"י הפרמטרים הבאים:

a. לגודל האוכלוסיה
b. להסתברות למוטציות
c. לאסטרטגיית הבחירה
d. לאסטרטגית השרידות (ELITSM ,AGING)
e. לאסטרטגית השיחלוף והמוטציה

We ran the code for different pop_size, mutation_rates, selction_methods, survival_strategies, exchange_operators and printed the results.