

AI LAB 2

פייסל סעדיה 208336321

אחמד גבארין 314722307

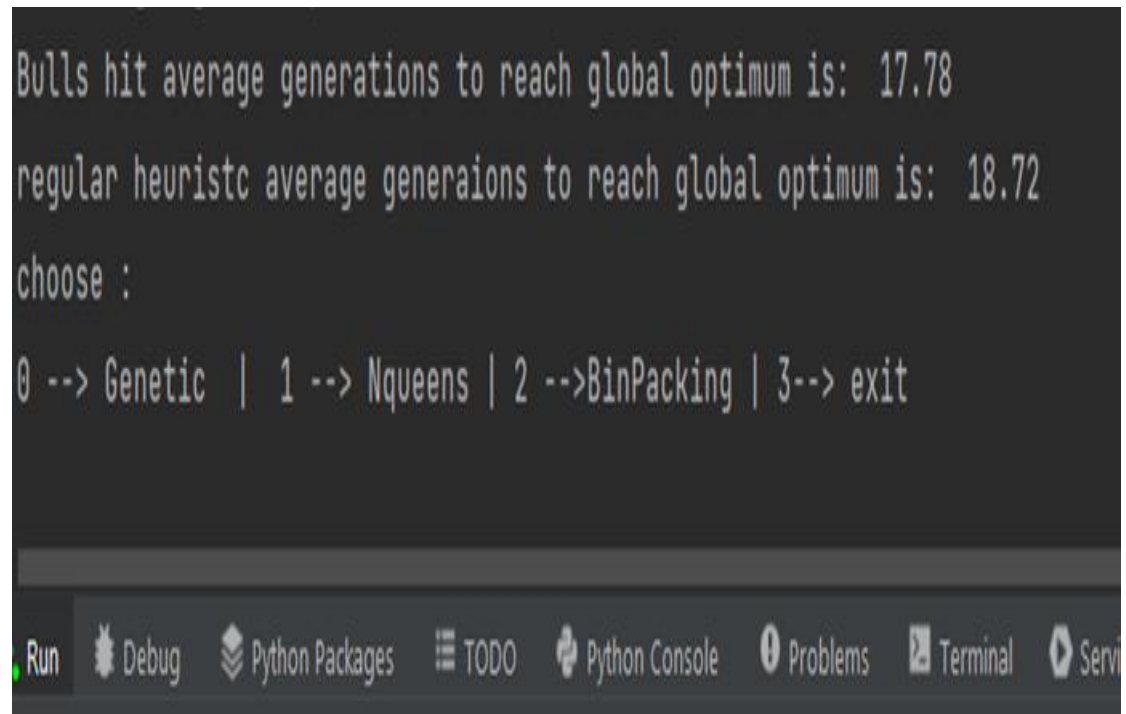
שאלה 1:

ייצגנו את מחרוזת המטרה בבינארי וכן גם
האוכלוסייה כך:

```
#Encode the target string in binary form
target = ''.join(format(ord(c), '08b') for c in "Hello, world!")
num_genes = len(target)
faisalsadi
```

עבור 50 הרצות של האלגוריתם המקורי מול
הבול פגיעה קיבלנו שיפור במספר האיטרציות
הממוצע עד לקבלת אופטימום גלובלי

```
Bulls hit average generations to reach global optimum is: 17.78
regular heuristic average generations to reach global optimum is: 18.72
choose :
0 --> Genetic | 1 --> Nqueens | 2 --> BinPacking | 3--> exit
```



שאלה 2:

הוספנו את המימושים של מטריקות הדימיון :

ming , edit distance , Kendal-tau

hamming distance, distance

להלן המימושים:

```

new *
def ming_distance(str1, str2):
    m = len(str1)
    n = len(str2)
    dp = [[0 for x in range(n + 1)] for x in range(m + 1)]

    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0:
                dp[i][j] = j
            elif j == 0:
                dp[i][j] = i
            elif str1[i - 1] == str2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(dp[i][j - 1], dp[i - 1][j], dp[i - 1][j - 1])

    return dp[m][n]

```

```

faisalsadi
def kendall_tau_distance(p, q):
    """
    Calculates the Kendall Tau distance between two permutations p and q
    """
    n = len(p)
    assert len(q) == n, "Permutations must be of equal length"

    inv_p = {v: k for k, v in enumerate(p)}
    inv_q = {v: k for k, v in enumerate(q)}

    # calculate the number of discordant pairs between the two permutations
    count = 0
    for i in range(n):
        for j in range(i + 1, n):
            if (p[i] < p[j] and q[inv_p[p[i]]] > q[inv_p[p[j]]]) or (p[i] > p[j] and q[inv_p[p[i]]] < q[inv_p[p[j]]]):
                count += 1

    return count

```

```

new *
def edit_distance(str1, str2):
    m = len(str1)
    n = len(str2)
    dp = [[0 for x in range(n + 1)] for x in range(m + 1)]

    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0:
                dp[i][j] = j
            elif j == 0:
                dp[i][j] = i
            elif str1[i-1] == str2[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = 1 + min(dp[i][j-1], dp[i-1][j], dp[i-1][j-1], dp[i-2][j-2] if i > 1 and j > 1 and str1[i-2] == str2[j-1]
                                   and str1[i-1] == str2[j-2] else float('inf'))

    return dp[m][n]

```

עבור בעיית הבול פגיעה בחרנו ב hamming
: distance

```

faisalsadi *
def hamming_distance(string1, string2 ):
    dist_counter = 0
    for i in range(len(string1)):
        if string1[i] != string2[i]:
            dist_counter += 1
    return dist_counter

```

שבה סופרים את מספר האינדקסים שבהם
אינם שווים

עבור בעיית ה NQueens ו Binning בחרנו ב kendall-tau לחישוב המרחק בין הפרמוטציות

שאלה 3:

2-עבור הסוג השני של המוטציה Non-Uniform Mutation אנחנו מטפלים בזה ב
תנאי השני כאשר אנו מחסירים באופן הדרגתי
את ההסתברות למוטציה.

3-עבור הסוג השלישי Adaptive Mutation אנחנו מכפילים בכל דור את אחוז המוטציה ב
top average selection ratio

```
if mutationtype == 2:  
    mutation_rate -= 0.03  
if mutationtype == 3:  
    mutation_rate *= tsp
```

-עבור הסוג הרביעי THM – Triggered Hyper Mutation אנחנו סופרים את מספר הדורות
שבהם הפיטנס לא השתפר ואז כשמגיעים
לסף מסויים אנו מפעילים מוטציה בהסתברות
מאוד גבוהה למשך כמה דורות בודדים

```

# Check if the best fitness has improved
if max(fitnesses) > last_best:
    last_best = fitnesses[0]
    best_solution = fitnesses[0]
    no_improvement_count = 0
else:
    no_improvement_count += 1
# Check if the trigger condition is met
if mutationtype==4:
    if no_improvement_count >= trigger_condition:
        # Apply hypermutation for the specified period
        for i in range(hypermutation_period):
            # Apply mutation with the high mutation rate
            population = apply_mutation(population, hypermutation_rate, pop_size)

```

5-עבור הסוג החמישי הוספנו שדה של
הסתברות מוטציה לכל גן ואז בכל דור אנו
מפעילים את פונקציית העדכון על כל הגנים
באוכלוסייה .

```

class Gene:
    # faisalsadi *
    def __init__(self, str, fitness, age):
        self.str = str
        self.fitness = fitness
        self.age = age
        self.rank = None
        self.mut_rate=0.3
    new *
    def update_mutationrate(self):
        self.mut_rate=0.3*(1-self.fitness/400)

```

שאלה 4 :

- a. בכל השיטות קיבלנו לרוב גיוון גנטי
זהה חוץ מהשיטה האחרונה שהיא ה self
adaptive שאר השיטות קיבלנו תוצאות
קרובות למעט ה THM שבו כמה פעמים
בודדות קיבלנו גיוון נמוך (כלמור
התכנסות למקסימום לוקאלי)
b. השיטה הכי מהירה הייתה השיטה
הבסיסית שאר השיטות היו קרובות אליה
להתכנסות למקסימום גלובלי חוץ מ ה
self adaptive שבו רוב הזמן קיבלנו
שהאלגוריתם נתקע במקסימום לוקאלי
ונשאר בו עד להשלמת מספר הדורות
המקסימלי.

שאלה 5:

NICHING אלגו:

הוספנו אותו בתוך פונקצית הפיטנס.

INDIVIDUALS יקבלו בונוסים על היותם שונים.

יותר שוני == יותר בונוסים, אינדודואלס דומים לא מקבלים כלום.

```
def fitness_bins(gene, population):
    bins = [[] for i in range(num_bins)]
    for i, j in enumerate(gene):
        bins[j].append(item_sizes[i])
    bins_used = sum(len(bin) > 0 for bin in bins)
    unused = sum(max(0, bin_size - sum(bin)) for bin in bins)

    # Niching function - give bonus points for being different
    niche_bonus = 0
    if(True):
        similarity_scores = [sum(gene[i] == other_gene[i] for i in range(len(gene))) for other_gene in population]
        similarity_scores.remove(len(gene)) # exclude self-similarity
        niche_count = sum(score < niche_radius for score in similarity_scores)
        niche_bonus = niche_count * niche_size
    return bins_used + unused + niche_bonus
```

CROWDING :

הוספנו אותה אחרי שלב של ה-CROSSOVER-


```

# mate
offspring = []
for parent1, parent2 in parents:
    if random.random() < crossover_rate:
        crossover_point = random.randint(1, len(item_sizes) - 1)
        child1 = parent1[:crossover_point] + parent2[crossover_point:]
        child2 = parent2[:crossover_point] + parent1[crossover_point:]
        ### NDC Step goes here ###
        if fitness_bins(child1) > fitness_bins(parent1) and \
           fitness_bins(child1) > fitness_bins(child2):
            offspring.append(child1)
        elif fitness_bins(child2) > fitness_bins(parent2) and \
            fitness_bins(child2) > fitness_bins(child1):
            offspring.append(child2)
        else:
            offspring.append(random.choice([parent1, parent2]))
        ### End of NDC ###
    else:
        offspring.append(random.choice([parent1, parent2]))

# mutate
for i in range(len(offspring)):
    for j in range(len(item_sizes)):
        if random.random() < mutation_rate:
            offspring[i][j] = random.randint(0, num_bins-1)

# select from population
population = random.choices(offspring, k=population_size)

Select best solution
best_gene = min(population, key=fitness_bins)
best_fitness = fitness_bins(best_gene)
print("Best solution:", best_gene)
print("Such that, each item is mapped to the container of the same index")
print("Best fitness:", best_fitness)

```

SPECIATION:

CLUSTERING עם אלגוריתם של KMEANS

```
# def clustering(population):
#     kmeans = KMeans(n_clusters=len(population)//30)
#     kmeans.fit(population)
#     labels = kmeans.labels_
#     clusters = [[] for i in range(len(set(labels)))]
#     for i in range(len(population)):
#         clusters[labels[i]].append(population[i])
#     return clusters

def clustering(population):
    max_clusters = len(population) // 30
    scores = []
    for n_clusters in range(2, max_clusters+1):
        kmeans = KMeans(n_clusters=n_clusters)
        kmeans.fit(population)
        labels = kmeans.labels_
        score = silhouette_score(population, labels)
        scores.append(score)
    best_n_clusters = np.argmax(scores) + 2 # add 2 to get ac
    kmeans = KMeans(n_clusters=best_n_clusters)
    kmeans.fit(population)
    labels = kmeans.labels_
    clusters = [[] for i in range(best_n_clusters)]
    for i in range(len(population)):
        clusters[labels[i]].append(population[i])
    return clusters
```

שילוב בתוך פונקציה: EVOLVE

```

def evolve():
    population = population_init(population_size)
    for generation in range(max_generations):
        # calculate fitness
        fitness_scores = [fitness_bins(gene) for gene in population]
        # Speciation with clustering
        clusters = clustering(population)
        speciated_population = []
        for cluster in clusters:
            speciated_population.extend(cluster)
        # Select parents
        parents = []
        for i in range(population_size):
            fitness_probs = [1 / (score + 1) for score in fitness_scores]
            parent1, parent2 = random.choices(speciated_population, weights=fitness_probs, k=2)
            parents.append((parent1, parent2))
        # mate
        offspring = []
        for parent1, parent2 in parents:
            if random.random() < crossover_rate:
                crossover_point = random.randint(1, len(item_sizes) - 1)
                offspring.append(parent1[:crossover_point] + parent2[crossover_point:])
                offspring.append(parent2[:crossover_point] + parent1[crossover_point:])
            else:
                offspring.append(parent1)
                offspring.append(parent2)
        # mutate
        for i in range(len(offspring)):
            for j in range(len(item_sizes)):
                if random.random() < mutation_rate:
                    offspring[i][j] = random.randint(0, num_bins-1)
        # select from population
        population = random.choices(speciated_population + offspring, k=population_size)
    # Select best solution
    best_gene = min(speciated_population, key=fitness_bins)
    best_fitness = fitness_bins(best_gene)
    print("Best solution:", best_gene)
    print("Such that, each item is mapped to the container of the same index")
    print("Best fitness:", best_fitness)

```

שאלה 6:

לפני הוספת האלגוריתמים:

```
→ AILab2023 git:(main) python3.10 Lab1/BinPackaging.py
Best solution: [2, 0, 0, 4, 4, 4, 2, 1, 3, 1]
Such that, each item is mapped to the container of the same index
Best fitness: 15
Took: 159.82937812805176 mseconds
```

עם:NICHING

```
→ AILab2023 git:(main) python3.10 Lab2/BinPackagingNiche.py
Best solution: [0, 1, 2, 1, 4, 4, 1]
Such that, each item is mapped to the container of the same index
Best fitness: 53
Took: 939.7501945495605 mseconds
```

קיבלנו פתרון יותר איכותי (הטרייד אופ זה שהוא לקח יותר זמן)

עם:CROWDING

```
→ AILab2023 git:(main) python3.10 Lab2/BinPackagingNDC.py
Best solution: [0, 4, 1, 4, 1, 3, 2, 1, 0, 2]
Such that, each item is mapped to the container of the same index
Best fitness: 16
```

גם איכות הפתרון השתפר

עם:Speciation

```
→ AILab2023 git:(main) python3.10 Lab2/BinPackagingSpeciation.py
Best solution: [2, 1, 3, 2, 1, 0, 4, 2, 1, 3]
Such that, each item is mapped to the container of the same index
Best fitness: 15
```

קיבלנו אותו פתרון מקודם אבל פה מאפשר
לנו להריץ תהליכים במקביל (כל CLUSTER
לבד) לכן במחשבים עם מספר תאים במעבד
זה עוזר לחסוך המון זמן בריצה. שילוב של
SPECIATION עם NICHING למשל מאפטם את
הפתרונות וגם ירוץ מהר יותר.
לקחנו בחשבון את:

- זמן הריצה
- ערך הפיטנס (יותר גדול == יותר טוב)
- ממוצע הפיטנס (יותר גדול == יותר טוב)

שאלה 7:

שתי פונקציות פיטנס f ו- g :

```

def f(x, y):
    # (x-5)**2 + (y-5)**2
    return x**2 + y**2

def g(x, y):
    return (x-5)**2 + (y-5)**2

def is_valid_f(x, y, radius):
    return x**2 + y**2 <= radius**2

def is_valid_g(x, y, radius):
    return (x-5)**2 + (y-5)**2 <= radius**2

# Generate an initial population of size N and radius R

def initialize_population(size, radius, func):
    population = []
    while len(population) < size:
        x = random.uniform(-radius, radius)
        y = random.uniform(-radius, radius)
        if func(x, y, radius):
            population.append((x, y))
    return population

```

האלגו expatation (במקום פונקצית ה-
evolve):

```

def exaptation(N, R, k, mu, sigma, max_generations):
    # Initialize populations
    population_f = initialize_population(N, R, is_valid_f)
    population_g = initialize_population(N, R, is_valid_g)

    for generation in range(max_generations):
        # Evaluate fitness
        fitness_scores_f = evaluate_population(population_f, f)
        fitness_scores_g = evaluate_population(population_g, g)

        # Select individuals
        selected_f = [tournament_selection(
            population_f, fitness_scores_f, k) for i in range(mu)]
        selected_g = [tournament_selection(
            population_g, fitness_scores_g, k) for i in range(mu)]

        # Generate offsprings
        offspring_f = []
        offspring_g = []
        for i in range(mu // 2):
            parent1_f = random.choice(selected_f)
            parent2_f = random.choice(selected_f)
            parent1_g = random.choice(selected_g)
            parent2_g = random.choice(selected_g)

            offspring_f.append(singlePointCrossover(parent1_f, parent2_f))
            offspring_f.append(singlePointCrossover(parent2_f, parent1_f))
            offspring_g.append(singlePointCrossover(parent1_g, parent2_g))
            offspring_g.append(singlePointCrossover(parent2_g, parent1_g))

        # Mutate offsprings
        mutated_f = [mutation(individual, sigma) for individual in offspring_f]
        mutated_g = [mutation(individual, sigma) for individual in offspring_g]

        # Merge populations
        population_f = selected_f + mutated_f
        population_g = selected_g + mutated_g

        # Immigrants between two islands
        Immigrants1 = random.sample(population_f, 10)
        Immigrants2 = random.sample(population_g, 10)

        Immigrants1 = list(filter(lambda x: is_valid_g(*x, R2), Immigrants1))
        Immigrants2 = list(filter(lambda x: is_valid_f(*x, R1), Immigrants2))

        weakest_indices_f = sorted(range(len(fitness_scores_f)), key=lambda k: fitness_scores_f[k])[:len(Immigrants2)]
        weakest_indices_g = sorted(range(len(fitness_scores_g)), key=lambda k: fitness_scores_g[k])[:len(Immigrants1)]

        # Replace weakest individuals with immigrant
        for i in weakest_indices_f:
            population_f[i] = Immigrants2.pop()
            population_g[i] = Immigrants1.pop()

    # Find the best solution for each function
    best_individual_f = max(population_f, key=lambda x: f(*x))
    best_individual_g = max(population_g, key=lambda x: g(*x))
    print("Best solution for f:", best_individual_f)
    print("Best fitness for f:", f(*best_individual_f))
    print("Best solution for g:", best_individual_g)
    print("Best fitness for g:", g(*best_individual_g))

```

כל פונקציה מאופטמת בא"י בנפרד.

בנוסף ממשנו הגירה Migration בין שני

האיים:

בוחרים אינדודאלים באופן שרירותי מאחד
האיים, אלה שעוברים את `is_valid` מחליפים
אינדודאלים הכי חלשים באי השני.

8.ע"י מימוש EXPATATION כך הפתרונות
שנלקחים בחשבון הם אלה שמקיימים תנאים
מסויימים תחת `structure` מסויים וזה מונע
שאינדודואלים לא מתאימים שייכנסו בתוך
האוכלסייה.

מימוש הגירה מאפשר להחליף אינדודואלים
חלשים עם אחרים טובים יותר עוזר בכך
שהוא מגדיל את מרחב הפתרון ומונע שאיפה
מוקדמת.

