AI LAB 1-2

פייסל סעדיה 208336321

אחמד גבארין 314722307

1) הוספנו קובץ parents selection שבו מימשנו את הפונקציות .

(a

```
# Roulette Wheel Selection Algorithm
Ahmed Jabareen
def roulette_wheel_selection(fitness_values):
    total_fitness = sum(fitness_values)
    probabilities = [fitness / total_fitness for fitness in fitness_values]
    cumulative_probabilities = [sum(probabilities[:i+1]) for i in range(len(probabilities))]
    r = random.random()
    for i in range(len(cumulative_probabilities)):
        if r < cumulative_probabilities[i]:
            return i

# RWS with scaled fitness values
Ahmed Jabareen
def roulette_wheel_selection_with_scaling(fitness_values):
    """Performs roulette wheel selection with scaling on a list of fitness values."""
    scaled_fitness_values = [fitness**2 for fitness in fitness_values]
    total_scaled_fitness = sum(scaled_fitness_values)
    probabilities = [fitness / total_scaled_fitness for fitness in scaled_fitness_values]
    cumulative_probabilities = [sum(probabilities[:i+1]) for i in range(len(probabilities))]
    r = random.random()
    for i in range(len(cumulative_probabilities)):
        if r < cumulative_probabilities[i]:
            return i
```

(b

```python
# SUS algorithm
Ahmed Jabareen
def stochastic_universal_sampling(fitness_values, num_samples):
    total_fitness = sum(fitness_values)
    fitness_per_sample = total_fitness / num_samples
    start = random.uniform(0, fitness_per_sample)
    pointers = [start + i * fitness_per_sample for i in range(num_samples)]
    selected_indices = []
    for pointer in pointers:
        i = 0
        while sum(fitness_values[:i+1]) < pointer:
            i += 1
        selected_indices.append(i)
    return selected_indices
```

```python
# SUS with scaling
Ahmed Jabareen
def stochastic_universal_sampling_with_scaling(fitness_values, num_samples):
    scaled_fitness_values = [fitness**2 for fitness in fitness_values]
    total_scaled_fitness = sum(scaled_fitness_values)
    fitness_per_sample = total_scaled_fitness / num_samples
    start = random.uniform(0, fitness_per_sample)
    pointers = [start + i * fitness_per_sample for i in range(num_samples)]
    selected_indices = []
    for pointer in pointers:
        i = 0
        cumulative_fitness = scaled_fitness_values[i]
        while cumulative_fitness < pointer:
            i += 1
            cumulative_fitness += scaled_fitness_values[i]
        selected_indices.append(i)
    return selected_indices
```

(C

```
Ahmed Jabareen
def tournament_selection(population, k):
    tournament = random.sample(population, k)
    winner = max(tournament, key=lambda x: x.fitness)
    return winner


Ahmed Jabareen
def tournament_selection_with_rank(population, k):
    ranked_population = sorted(population, key=lambda x: x.fitness, reverse=True)
    for i in range(len(ranked_population)):
        ranked_population[i].rank = i+1
    tournament = random.sample(ranked_population, tournament_size)
    winner = max(tournament, key=lambda x: x.rank)
    return winner
```

2)הוספנו שדה חדש ל- gene.py שזה. age

כך שב elitism-נותנים פחות סיכויים לגינומים שמזדקנים.

(3

a) ייצוג :מחרוזת באורך N תייצג לוח באורך N*N כך שבעזרת האינדקס והמספר שנמצא בתוך האינדקס נקבל את הקורדינאטה של המלכה ה i למשל :

[1,2,3,4,5,6,7] המלכה הראשונה נמצאת ב (1,1) השנייה ב (2,2) וכן הלאה ..

(b

Pmx : מגרילים אינדיקס ואז מחליפים את האותיות בהורים

```
faisalsadi *
def pmx(self):
    for i in range(int(self.numElitation), self.GA_POPSIZE):
        i1=0
        i2=0
        fitnessses = [obj.fitness for obj in self.population]
        if self.Parent_selection == 0:
            i1 = randint(0, self.GA_POPSIZE / 2)
            i2 = randint(0, self.GA_POPSIZE / 2)
        if self.Parent_selection==1:
            i1 = ParentsSelection.roulette_wheel_selection(fitnessses)
            i2 = ParentsSelection.roulette_wheel_selection(fitnessses)
        if self.Parent_selection == 2:
            ind = ParentsSelection.stochastic_universal_sampling(fitnessses,self.GA_POPSIZE)
            i1 = ind[0]
            i2 = ind[1]
        i3 = randint(0, self.N-1)
        secondChar = self.population[i2].NQueens[i3]
```

```
        for j in range(1, self.N):
            self.nextPopulation[i].NQueens[j] = self.population[i1].NQueens[j]
        for j in range(self.N):
            if self.population[i1].NQueens[j] == secondChar:
                self.nextPopulation[i].NQueens[j] = self.population[i1].NQueens[i3]
                self.nextPopulation[i].NQueens[i3] = self.population[i1].NQueens[j]
                break
        if self.mutateType == 0:
            if random() < self.GA_MUTATION:
                self.inversion_mutation(i)
        else:
            if random() < self.GA_MUTATION:
                self.random_mutation(i)
```

: CX

```
faisalsadi *
def cx(self):
    for i in range(int(self.numElitation), self.GA_POPSIZE):
        i1 = randint(0, self.GA_POPSIZE-1)
        i2 = randint(0, self.GA_POPSIZE-1)
        parent1 = self.population[i1].NQueens
        parent2 = self.population[i2].NQueens
        indicesArray = []
        child = []
        odd = False
        while len(indicesArray) < self.N:
            firstIndex = self.findFirstIndex(indicesArray)
            index = firstIndex
            while True and firstIndex < self.N:
                indicesArray.append(index)
                if odd:
                    child.append(parent1[index])
                    index = parent1.index(parent2[index])
                else:
                    child.append(parent2[index])
                    index = parent1.index(parent2[index])
```

```
                index = parent1.index(parent2[index])
                if index == firstIndex:
                    break
            odd = not odd
        self.nextPopulation[i].NQueens = child
        if self.mutateType == 0:
            if random() < self.GA_MUTATION:
                self.inversion_mutation(i)
        else:
            if random() < self.GA_MUTATION:
                self.random_mutation(i)
```

(C

```
faisalsadi *
def inversion_mutation(self,i):
        i1 = randint(0, self.N - 3)
        i2 = randint(i1 + 1, self.N - 2)
        i3 = randint(i2, self.N - 1)
        self.population[i].NQueens = self.population[i].NQueens[0:i1] + self.population[i].NQueens[i2:i3] + self.population[i].NQueens[i1:i2][::-1

faisalsadi *
def Shuffle_mutation(self,i):
        i1 = randint(0, self.N - 3)
        i2 = randint(i1 + 1, self.N - 2)
        str = self.population[i].NQueens[i1:i2]
        shuffle(str)
        self.population[i].NQueens = self.population[i].NQueens[0:i1] + str + self.population[i].NQueens[i2:]
```

(D

מימשנו פונקציית פיטנס שמסתמכת על מספר ההתנגשיות בפרמוטציה מסויימת כך שאם יש הרבה התנגשויות אז יש מקבלים קנס יותר גבוה , עוברים על מערך המלכות ומחשבים את מספר ההתנגשויות (מחלקים ב 2 בגלל הסימטריה " מלכה I מאיימת על J זה אותו דבר כמו שמלכה J תאיים על I ")

```
faisalsadi *
def calc_conflict(self, NQueens, j):
    conflicts = 0
    row = NQueens[j]
    col = j
    for i, k in zip(range(row), range(col)):
        if NQueens[k] == i:
            conflicts += 1

    for i, k in zip(range(row + 1, self.N), range(col)):
        if NQueens[col - 1 - k] == i:
            conflicts += 1

    for i, k in zip(range(row), range(col + 1, self.N)):
        if NQueens[k] == row - 1 - i:
            conflicts += 1

    for i, k in zip(range(row + 1, self.N), range(col + 1, self.N)):
        if NQueens[k] == i:
            conflicts += 1

    for i in range(self.N):
        if NQueens[i] == row and i != col:
            conflicts += 1
    return conflicts
```

```
faisalsadi *
def calc_fitness(self):
    for i in range(self.GA_POPSIZE):
        fitness = 0
        for j in range(self.N):
            fitness += self.calc_conflict(self.population[i].NQueens, j)
        self.population[i].fitness = fitness / 2
```

ודוגמה להרצה עבור שיטת שחלוף PMX ומוטציית
ערבול ובשיטת בחירת הורים Naïve Parent Selection

```
Best Gene in generation  0  : 1 2 5 7 4 0 3 6  , Fitness : 1.0
Best Gene in generation  1  : 1 7 4 0 5 2 6 3  , Fitness : 1.0
Best Gene in generation  2  : 1 2 5 7 4 0 3 6  , Fitness : 1.0
Best Gene in generation  3  : 1 7 4 0 5 2 6 3  , Fitness : 1.0
Best Gene in generation  4  : 1 2 5 7 4 0 3 6  , Fitness : 1.0
Best Gene in generation  5  : 1 7 4 0 5 2 6 3  , Fitness : 1.0
Best Gene in generation  6  : 1 2 5 7 4 0 3 6  , Fitness : 1.0
Best Gene in generation  7  : 1 7 4 0 5 2 6 3  , Fitness : 1.0
Best Gene in generation  8  : 1 2 5 7 4 0 3 6  , Fitness : 1.0
Best Gene in generation  9  : 1 7 4 0 5 2 6 3  , Fitness : 1.0
Best Gene in generation  10  : 1 2 5 7 4 0 3 6  , Fitness : 1.0
Best Gene in generation  11  : 5 3 6 0 2 4 1 7  , Fitness : 0.0
```

4) הוספנו קובץ BinPackaging.py שמטפל בבעית ה-
Bin Packaging.

```python
import random


# Define chromosome and fitness function
def create_genes():
    return [random.randint(0, num_bins-1) for i in range(len(item_sizes))]


def fitness_bins(gene):
    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)
    return bins_used + unused

def population_init(population_size):
     return [create_genes() for i in range(population_size)]


# Evolve population
def evolve():
    population = population_init(population_size)
    for generation in range(max_generations):
        # calculate fitness
        fitness_scores = [fitness_bins(gene) for gene in population]
        # Select parents
        parents = []
        for i in range(population_size):
            fitness_probs = [1 / (score + 1) for score in fitness_scores]
            parent1, parent2 = random.choices(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(population + offspring, k=population_size)
    # Select best solution
```

```python
    # 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)

f __name__ == "__main__":
    # GA parameters
    population_size = 100
    max_generations = 100
    mutation_rate = 0.1
    crossover_rate = 0.9
    num_bins = 5

    # problem parameters
    bin_size = 10
    item_sizes = [3, 4, 5, 2, 1, 7, 6]

    evolve()
```

a. פונקצית הפיטנס שמימשנו:

```python
def fitness_bins(gene):
    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)
    return bins_used + unused
```

היא לוקחת גינום כקלט אשר מיוצר על ידי הפונקציה:

```python
def create_genes():
    return [random.randint(0, num_bins-1) for i in range(len(item_sizes))]
```

כך שכול גינום הוא מהפורמט הבא:

למשל: [4, 3, 2, 2, 0, 3, 0]

החפץ באינדקס 0 הולך למכולת בעלת אינדקס 0

החפץ באינדקס 1 הולך למכולת בעלת אינדקס 3

החפץ באינדקס 3 הולך למכולת בעלת אינדקס 2

החפץ באינדקס 4 הולך למכולת בעלת אינדקס 2

וכו

הפונקציה אחר כך מסדרת אותם לתוך bins

הערך של הפיטנס: אנחנו מעוניינים בגינומים שמהווים פתרון כך שבו השתמשנו בפחות מכולות כך שכל אחת מהם תכיל כמה שיותר חפצים, לכן רוצים למקסם את הערך של bins_used + unused.

b. האלגוריתם החמדני רץ באופן מהיר יותר כתוצאה מכך שהוא לא עושה חישובים מסובכים ואינו משתמש בהרבה משאבים.
האלגוריתם הגינטי לוקח לו קצת יותר זמן (בסביבות 150-200 מל שניות) אבל נותן תוצאות יותר קרובות לפתרון האופטמלי.

5.

אנו מציגים 3 מדדים : טווח ערכי הפיטנס , שונות הפיטנס ו Top-Average Selection Probability Ratio

החישוב נעשה בפונקציה הזאת שמשתמשים בה בכל הבעיות הקודמות

```
Ahmed Jabareen *
def selection_pressure(fitness_scores,k,size):
    '''
    Measuring Selection Pressure:
    The ratio between the probability that the most fit is selected to the probability
    that the average member is selected.
    a. Fitness Variance
    b. Ratio as explained above
    * Per generation
    '''
    # Calculate selection pressure
    selection_pressure = max(fitness_scores) - min(fitness_scores)
    # Calculate fitness variance
    fitness_variance = np.var(fitness_scores)
    # Calculate Top-Average Selection Probability Ratio (TASPR)
    selection_probabilities = [fitness_score / sum(fitness_scores) for fitness_score in fitness_scores]
    sorted_selection_probabilities = sorted(selection_probabilities, reverse=True)
    top_selection_probabilities_sum = sum(sorted_selection_probabilities[:k])
    average_selection_probability = 1 / size
    taspr = top_selection_probabilities_sum / (k * average_selection_probability)

    print("--Selection Pressure: {}, Fitness Variance: {}, Top-Average Selection Probability Ratio: {}".format(selection_pressure,fitness_variance,

    pass
```

6.

אנו מציגים 2 מדדים : המרחק של הגנים ומספר האללים השונים

החישוב נעשה בפונקציה הזאת שמשתמשים בה בכל הבעיות הקודמות

פונקציית המרחק הייתה hamming distance שבה אנחנו מחזירים את מספר האינדיקסים שבהם שני גינים אינם תואמים

```
Ahmed Jabareen *
def genetic_diversification(population):
    '''
    * Hamming distanse between individuals
    * Number of different alleles in population
    * Per generation
    '''
    dist=0
    for str1 in population:
        for str2 in population:
            dist+=hamming_distance(str1,str2)
    num_different_alleles = len(set(map(tuple, population)))
    print("--number of different alleles: {}, Distance between genes: {}".format(
        num_different_alleles, dist))
    pass
```

הרצה והצגת הדיווח בכל דור עבור הסעיפים 5 ו 6 (עבור בעיית המלכות):

```
Best Gene in generation  1  : 2 5 7 4 1 3 0 6  , Fitness : 1.0
--Selection Pressure: 14.0, Fitness Variance: 6.021275, Top-Average Selection Probability Ratio: 2.0898641588296756
--number of different alleles: 100, Distance between genes: 69200
Best Gene in generation  2  : 4 0 7 3 1 6 2 5  , Fitness : 1.0
--Selection Pressure: 13.5, Fitness Variance: 6.447899999999999, Top-Average Selection Probability Ratio: 2.1063394683026586
--number of different alleles: 99, Distance between genes: 69000
Best Gene in generation  3  : 6 2 0 4 1 7 5 3  , Fitness : 0.5
--Selection Pressure: 13.0, Fitness Variance: 5.449400000000001, Top-Average Selection Probability Ratio: 2.108294930875576
--number of different alleles: 100, Distance between genes: 69140
Best Gene in generation  4  : 4 0 7 3 1 6 2 5  , Fitness : 1.0
--Selection Pressure: 9.5, Fitness Variance: 5.0908999999999995, Top-Average Selection Probability Ratio: 1.9639065817409764
--number of different alleles: 100, Distance between genes: 68756
Best Gene in generation  5  : 6 2 0 4 1 7 5 3  , Fitness : 0.5
--Selection Pressure: 15.0, Fitness Variance: 6.324400000000001, Top-Average Selection Probability Ratio: 2.246543778801843
--number of different alleles: 100, Distance between genes: 68706
Best Gene in generation  6  : 4 0 7 3 1 6 2 5  , Fitness : 1.0
--Selection Pressure: 10.5, Fitness Variance: 5.384599999999999, Top-Average Selection Probability Ratio: 2.042410714285714
--number of different alleles: 100, Distance between genes: 68506
Best Gene in generation  7  : 6 2 0 4 1 7 5 3  , Fitness : 0.5
--Selection Pressure: 12.0, Fitness Variance: 5.8724, Top-Average Selection Probability Ratio: 2.2641509433962264
--number of different alleles: 100, Distance between genes: 68612
Best Gene in generation  8  : 4 6 1 5 2 0 3 7  , Fitness : 0.0
--Selection Pressure: 9.0, Fitness Variance: 4.661275000000001, Top-Average Selection Probability Ratio: 2.01765447667087
--number of different alleles: 99, Distance between genes: 68608
choose :
0 --> Genetic  |  1 --> Nqueens | 2 -->BinPacking | 3--> exit
3
```

7.

a)

גודל האוכלוסיה: עבור שתי הבעיות היינו מקבלים פחות דורות אבל זמן ריצה יותר גדול עם הגדלת גודל האוכלוסיה

b)

הסתברות למוטציה: בבעית N המלכות אם מגדילים הסתברות למוטציה אז מספר הדורות וזמן הריצה שניהם משתפרים תמיד אבל binpacking כן השתפר ברוב המקרים אבל לפעמים זמן הריצה היה יותר גרוע בפחות משנייה אבל לרוב השתפר

c)

אסטרטגיית הבחירה בשני המקרים הטורניר זה הכי מהר וטוב שהיה

d)

אסטרטגיית השרידות:לפי AGING יותר טוב בשני המקרים

e)

לN המלכות CX היה יותר טוב ול binpacking שיטת ה Uniform הניבה תוצאות יותר טובות.

8)

עבור הבעיות הפרמטרים הכי טובים היו

Pop size=200

Elite rate=0.1

Mutation rate=0.3

a)

בדקנו עבור כל פרמטר את ההשפעה שלו על הגיוון והלחץ ושאפנו לכמה שיותר איזון בין הגיוון והלחץ וזה מה שהוביל אותנו לפרמטרזציה המיטבית

b)

שמירה על איזון בין גיוון ולחץ : כדי להגיע לתוצאות הכי טובות הפרמטרים שאנחנו מעבירים צריכים לשמור על האיזון , כלומר שלא יהיה גיוון גבוה מידי ואז אנחנו נהרוס מועמדים טובים לדור הבא ובאותו אופן שלא יהיה לחץ בחירה גבוה מאוד ואז נתקעים באופטימום לוקאלי למשך זמן רב.