מעבדה בבינה מלאכותית

סאמר חרעובה – 209050202

עבד אלרחמן אבו חוסין -208517631

חלק א

.1

```
def calc_fitness(self):
    mean = 0
    for i in range(self.pop_size):
        self.population[i].calculate_fittness(self.target, self.target_size, self.fitnesstype)
        # mean += self.population[i].fitness
    for i in range(self.pop_size):
        mean += self.population[i].fitness

    self.pop_mean = mean / self.pop_size
```

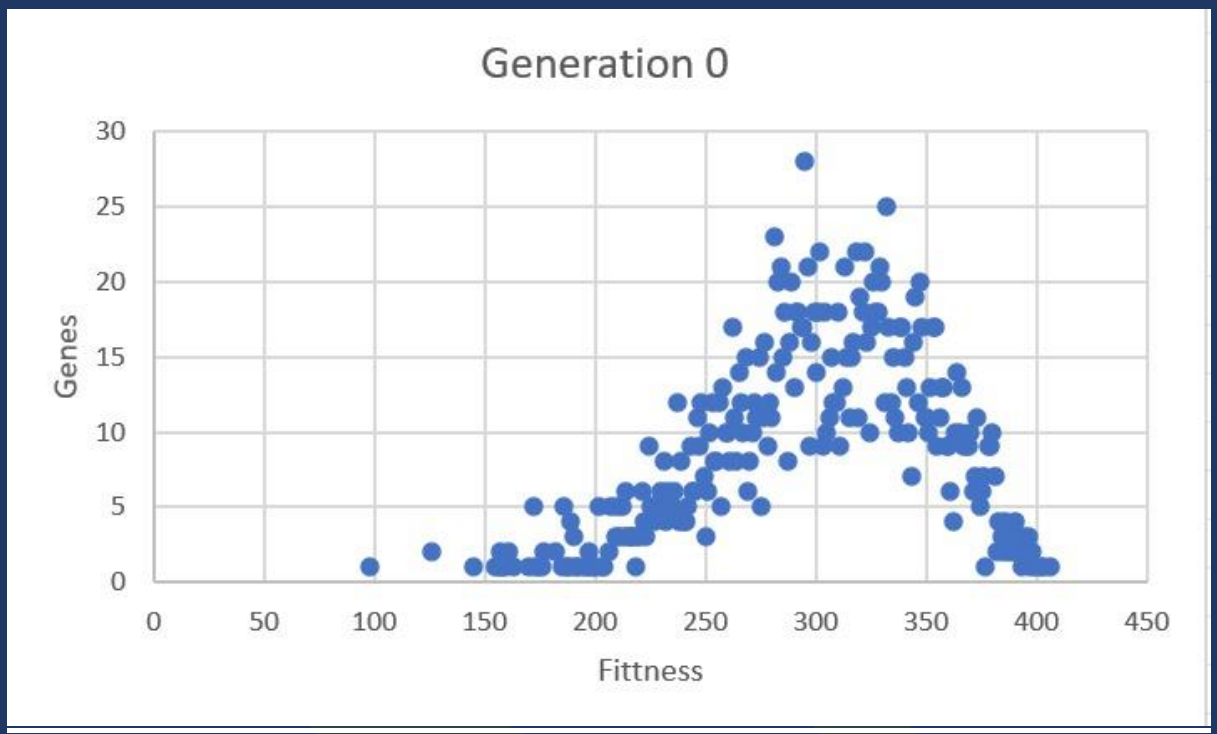
```
variance = lambda x: math.sqrt((x[0] - x[1]) ** 2)
```

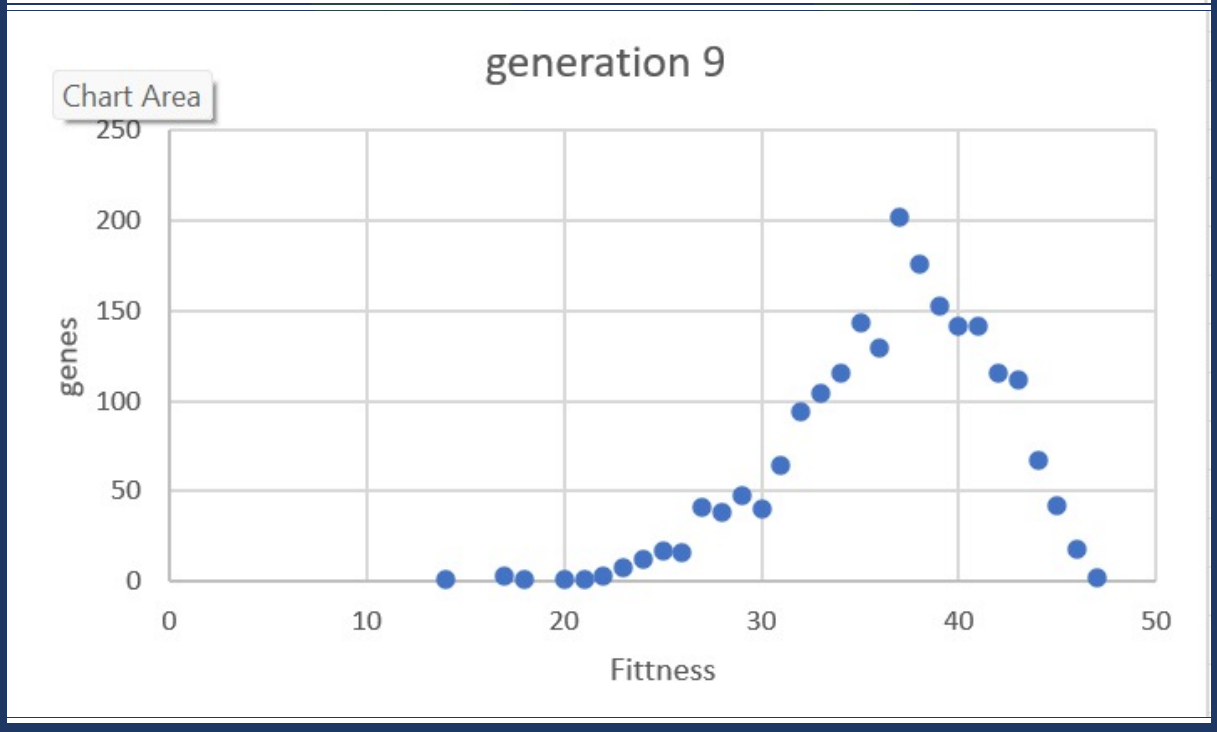
```
#  prints mean and variance
print_mean_var = lambda x: print(f"Mean: {x[0]} ,Variance: {x[1]}", end=" ")
# prints time
```

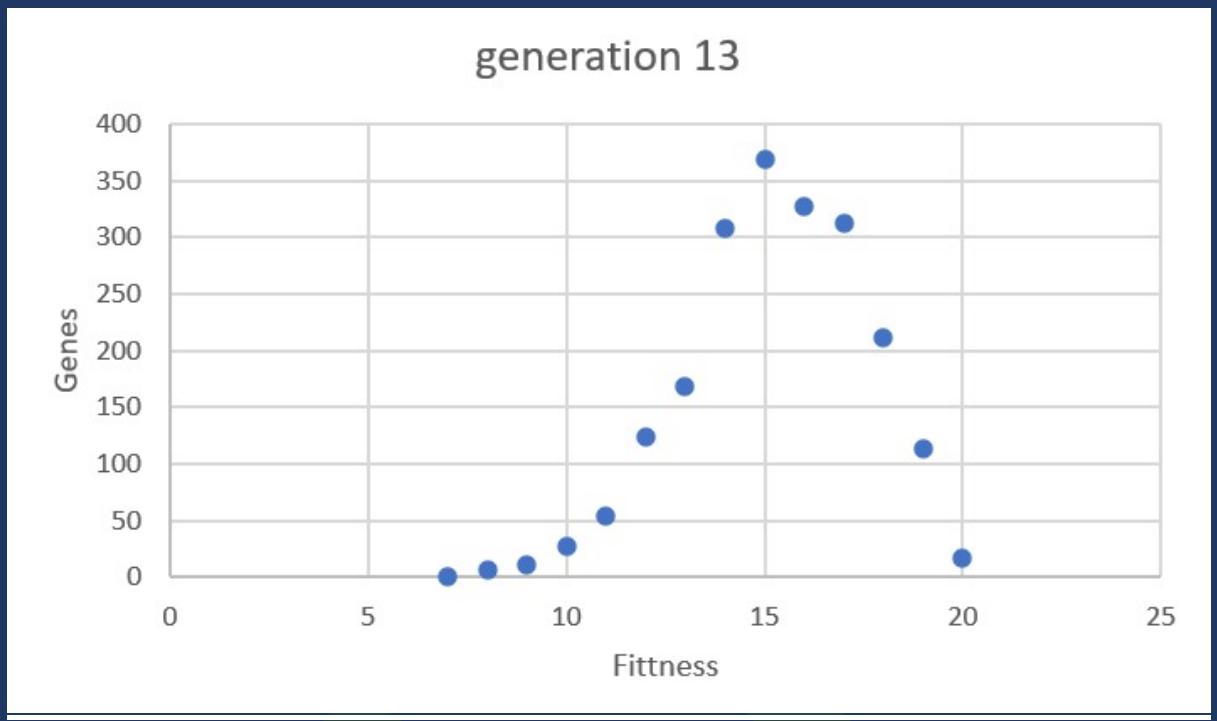
.2

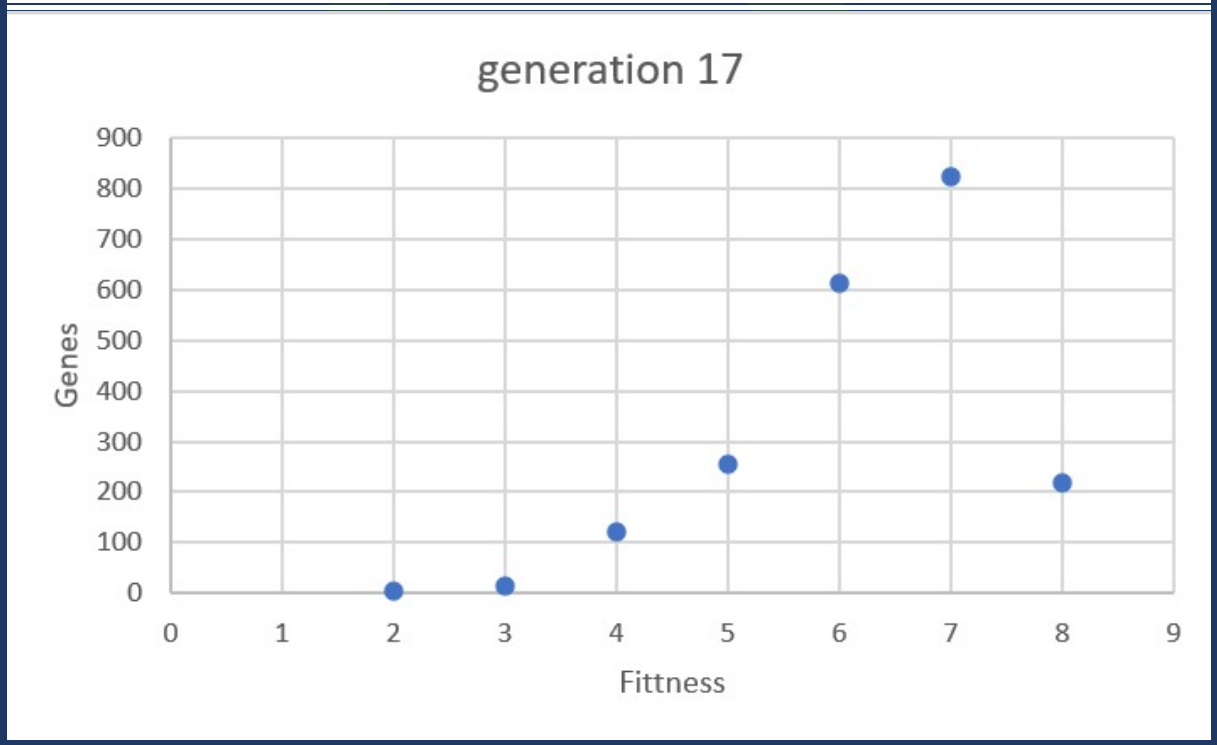
```
Best: ['h', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!'] ,fittness: 0  Mean: 82.734375 ,Variance: 82.734375 Time
```

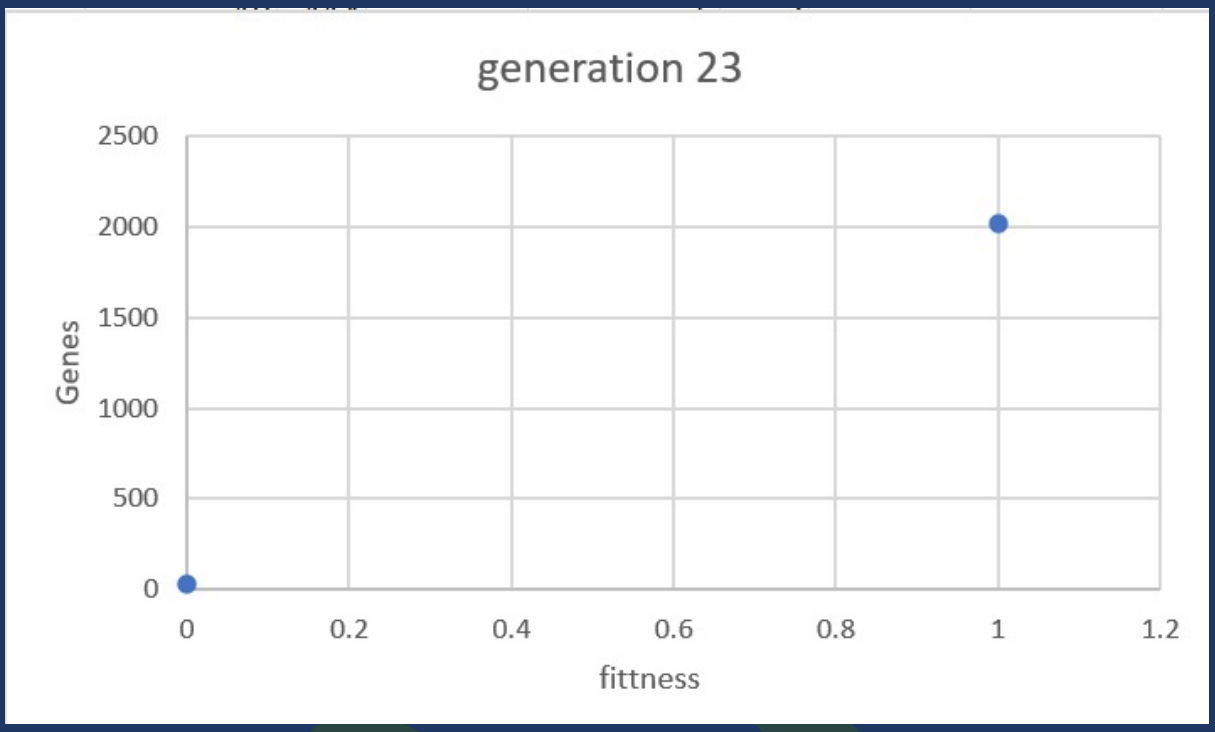
```
def handle_initial_time(self):
    self.tick = time.time()
    self.sol_time = time.perf_counter()

def handle_prints_time(self):
    runtime = time.perf_counter() - self.sol_time
    clockticks = time.time() - self.tick
    print_B(self.solution)
    print_mean_var((self.pop_mean, variance((self.pop_mean, self.solution.fitness))))
    print_time((runtime, clockticks))
```


Generation 0
Genes
Fittness
0
5
10
15
20
25
30
0
50
100
150
200
250
300
350
400
450


generation 9
Chart Area
genes
Fittness
0
50
100
150
200
250
0
10
20
30
40
50


generation 13
400
350
300
250
200
150
100
50
0
Genes
0
5
10
15
20
25
Fittness


generation 17
900
800
700
600
500
400
300
200
100
0
Genes
0
1
2
3
4
5
6
7
8
9
Fittness



Attached text file with all the data *

.4

יצרנו class cross_types חדש שמכניסים אותו כפרמטר לכל אלגוריתם גנרי שמוגדר ע"י class algorithems שמשתמשים בו לכל אלגוריתם שנרצה הלא.

```
class cross_types:

    def __init__(self):
        # maps functions to numbers so that we can choose which one to assign
        self.select = {CROSS1: self.one_cross, CROSS2: self.two_cross, UNI_CROSS: self.uniform_cross

    def one_cross(self, citizen1, citizen2):...

    def two_cross(self, citizen1, citizen2):...

    def uniform_cross(self, citizen1, citizen2):...
```

מימוש הפונקציות

```
def one_cross(self, citizen1, citizen2):
    target_size =min (len(citizen1.object),len(citizen2.object))
    spos = random.randint(0, target_size)
    return citizen1.object[0:spos] + citizen2.object[spos:target_size], citizen2.object[0:spos] + citizen1.object[spos:target_size]

def two_cross(self, citizen1, citizen2):
    target_size = min(len(citizen1.object), len(citizen2.object))
    spos = random.randint(0, target_size - 2)  # we need at least 3 portions
    spos2 = random.randint(spos,  target_size - 1)  # we need at least 2 portions
    first = citizen1.object[0:spos] + citizen2.object[spos:spos2] + citizen1.object[spos2:]
    sec = citizen2.object[0:spos] + citizen1.object[spos:spos2] + citizen2.object[spos2:]
    return first, sec

def uniform_cross(self, citizen1, citizen2):
    target_size = min(len(citizen1), len(citizen2))
    object1 = []
    object2 = []
    for i in range(target_size):
        if random.random() > 0.5:
            object1 = object1[:] + citizen2.object[i]
            object2 = object2[:] + citizen1.object[i]
        else:
            object1 = object1[:] + citizen1.object[i]
            object2 = object2[:] + citizen2.object[i]
    return object1, object2
```

.5

בשביל ה fitness יצרנו עוד class fitness_selector שבו יש את כל פונקציות ה fitness שמימשנו ,ובו נכניס כל פונקציית fitness שננמש הלא.

```python
class fitness_selector:
    def __init__(self):
        self.select = {0: self.distance_fittness, 1: self.bul_pqia

    def distance_fittness(self, object, target, target_size):
        fitness = 0
        for j in range(target_size):
            fit = ord(object[j]) - ord(target[j])
            fitness += abs(fit)
        return fitness

    def bul_pqia(self, object, target, target_size):
        fitness = 0
        for i in range(target_size):
            if ord(object[i]) != ord(target[i]):
                fitness += PENALTY if object[i] in target else HIGH_PENALTY
        return fitness
```

```python
PENALTY = 30
HIGH_PENALTY = 90
```

.6

לפי הפלט שקיבלנו (מוצג למטה) אנחנו רואים כי תמיד היוריסטיקה של "בול פגיעה" הייתה יותר טובה כי תמיד מייצרת פחות דורות ו מסיימת בפחות זמן.

האלגוריתם שלנו משאיר 10 אחוז מהאוכלוסייה כ איליתא ואלה יש להם ערך הפיטניס הקטן ביותר משאר האוכלוסייה .

כלומר המוטציה קורא ל 90% מהאוכלוסייה שערך הפיטניס שלהן גדול או שווה ל "איליתא " מה שמייצר ואריאציות ב אלה שהם לא "איליתא " וכיוון שבול פגיעה נותן הערכה יותר טובה , האוכלוסייה תתכנס יותר מהר לפתרון

One point crossover, with distance heuristic:

```
Best: ['d', 's', 'n', 'R', 'J', '4', '>', 'z', 'x', 'r', 'W', '5'] ,fittness: 184  Mean: 408.49560546875 ,Variance: 224.49560546875 Time :  0.12133949999999949  ticks: 0.12147808074951172
Best: ['v', 'T', 'l', 'P', 'z', '$', 'S', 'z', 'r', 'r', 'l', 'A'] ,fittness: 135  Mean: 307.62744140625 ,Variance: 172.62744140625 Time :  0.17153139999999922  ticks: 0.171534776687622 07
Best: ['c', 'R', 'n', 'r', 'c', '"', 'U', 'v', 'd', 'j', 'z', '4'] ,fittness: 112  Mean: 241.71337890625 ,Variance: 129.71337890625 Time :  0.2230635000000003  ticks: 0.222771167755126 95
Best: ['x', 'l', 'r', 'l', 'u', '#', 'X', 'I', 'l', 'n', 'c', '&'] ,fittness: 91  Mean: 192.99462890625 ,Variance: 101.99462890625 Time :  0.27202589999999915  ticks: 0.272010326385498 05
Best: ['r', '^', 'w', 'n', 's', '"', 'R', 'f', 'g', 'q', 'c', '$'] ,fittness: 70  Mean: 153.2744140625 ,Variance: 83.2744140625 Time :  0.30934419999999996  ticks: 0.3089621067047119
Best: ['f', '^', 'e', 'q', 'n', '"', 'S', 'p', 'r', 'r', 'l', ' '] ,fittness: 49  Mean: 122.4755859375 ,Variance: 73.4755859375 Time :  0.3605262000000007  ticks: 0.35996127128601074
Best: ['f', 'l', 'g', 't', 'k', ' ', 'T', 'l', 't', 's', 'c', '$'] ,fittness: 45  Mean: 98.7861328125 ,Variance: 53.7861328125 Time :  0.40890800000000027  ticks: 0.4087793827056885
Best: ['f', 'j', 'j', 'p', 'o', '"', 'R', 'f', 'r', 'b', 'b', '!'] ,fittness: 41  Mean: 80.58935546875 ,Variance: 39.58935546875 Time :  0.4822880000000005  ticks: 0.481689453125
Best: ['k', 'e', 'k', 'l', 'u', '"', 'U', 'u', 'q', 'r', 'd', '#'] ,fittness: 29  Mean: 65.9140625 ,Variance: 36.9140625 Time :  0.5593629  ticks: 0.558995246887207
Best: ['i', 'a', 'n', 'k', 'l', '#', 'S', 'p', 'n', 'l', 'e', '#'] ,fittness: 26  Mean: 54.75341796875 ,Variance: 28.75341796875 Time :  0.6193720999999996  ticks: 0.6196804046630859
Best: ['k', 'e', 'k', 'l', 'u', '"', 'U', 'j', 'r', 'n', 'd', '!'] ,fittness: 21  Mean: 46.322265625 ,Variance: 25.322265625 Time :  0.6755741000000004  ticks: 0.6756558418273926
Best: ['k', 'e', 'k', 'l', 'o', '$', 'S', 'p', 'r', 'l', 'd', '!'] ,fittness: 13  Mean: 39.96044921875 ,Variance: 26.96044921875 Time :  0.7317622000000004  ticks: 0.7317545413970947
Best: ['k', 'e', 'k', 'l', 'o', '$', 'S', 'p', 'r', 'l', 'd', '!'] ,fittness: 13  Mean: 34.3984375 ,Variance: 21.3984375 Time :  0.7829466000000007  ticks: 0.7830114364624023
Best: ['i', 'e', 'k', 'l', 'o', '$', 'S', 'p', 'r', 'l', 'd', '!'] ,fittness: 11  Mean: 30.154296875 ,Variance: 19.154296875 Time :  0.8269251000000004  ticks: 0.8266513347625732
Best: ['g', 'e', 'l', 'l', 'o', '$', 'U', 'p', 'r', 'j', 'd', '!'] ,fittness: 10  Mean: 26.486328125 ,Variance: 16.486328125 Time :  0.8924993000000008  ticks: 0.892045259475708
Best: ['i', 'e', 'l', 'l', 'o', ' ', 'S', 'o', 'r', 'l', 'd', '!'] ,fittness: 5  Mean: 22.095703125 ,Variance: 17.095703125 Time :  0.9608238999999994  ticks: 0.9610447883605957
Best: ['g', 'e', 'l', 'l', 'o', ' ', 'U', 'p', 'r', 'l', 'd', '!'] ,fittness: 4  Mean: 19.7109375 ,Variance: 15.7109375 Time :  1.0145005999999999  ticks: 1.0140705108642578
Best: ['g', 'e', 'l', 'l', 'o', ' ', 'U', 'p', 'r', 'l', 'd', '!'] ,fittness: 4  Mean: 16.5068359375 ,Variance: 12.5068359375 Time :  1.0577477000000002  ticks: 1.057769536972046
Best: ['g', 'e', 'l', 'l', 'o', ' ', 'V', 'o', 'r', 'l', 'd', ' '] ,fittness: 3  Mean: 15.09033203125 ,Variance: 12.09033203125 Time :  1.1073775000000001  ticks: 1.107527494430542
Best: ['g', 'e', 'l', 'l', 'o', ' ', 'W', 'p', 'r', 'l', 'd', '!'] ,fittness: 2  Mean: 13.83544921875 ,Variance: 11.83544921875 Time :  1.1678684000000004  ticks: 1.1681482791900635
Best: ['h', 'e', 'l', 'l', 'o', ' ', 'X', 'o', 'r', 'l', 'd', '!'] ,fittness: 1  Mean: 12.41552734375 ,Variance: 11.41552734375 Time :  1.2184539  ticks: 1.218073844909668
Best: ['h', 'e', 'l', 'l', 'o', ' ', 'X', 'o', 'r', 'l', 'd', '!'] ,fittness: 1  Mean: 12.65869140625 ,Variance: 11.65869140625 Time :  1.2587609999999998  ticks: 1.258937120437622
Best: ['h', 'e', 'l', 'l', 'o', ' ', 'X', 'o', 'r', 'l', 'd', '!'] ,fittness: 1  Mean: 10.60302734375 ,Variance: 9.60302734375 Time :  1.2998969999999996  ticks: 1.3000812530517578
Best: ['h', 'e', 'l', 'l', 'o', ' ', 'X', 'o', 'r', 'l', 'd', '!'] ,fittness: 1  Mean: 9.47265625 ,Variance: 8.47265625 Time :  1.3578396000000001  ticks: 1.358011960983276 4
Best: ['h', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!'] ,fittness: 0  Mean: 9.2568359375 ,Variance: 9.2568359375 Time :  1.4161525000000008  ticks: 1.4159109592437744
```

One point crossover, with Bulls and Cows Heuristic:

```
Best: ['C', 'l', 'd', 'o', 'o', '!', 'k', '.', 'o', '!', '3', '^'] ,fittness: 630  Mean: 1004.6923828125 ,Variance: 374.6923828125 Time :  0.09586030000000001  ticks: 0.09580731391906738
Best: ['C', 'l', 'd', 'o', 'o', '!', 'k', '.', 'o', '!', '3', '^'] ,fittness: 630  Mean: 988.7451171875 ,Variance: 278.7451171875 Time :  0.14736049999999956  ticks: 0.146763563156127 93
Best: ['C', 'l', 'd', 'o', 'o', '!', 'k', 'Z', 'o', '!', '9', '!'] ,fittness: 540  Mean: 817.8662109375 ,Variance: 277.8662109375 Time :  0.21762239999999977  ticks: 0.21678805351257324
Best: ['9', 'e', 's', 'Z', 'r', 'W', 'W', 'o', 'e', '5', 'o', 'l'] ,fittness: 510  Mean: 727.8369140625 ,Variance: 217.8369140625 Time :  0.2642793999999995  ticks: 0.2638130187988281
Best: ['C', 'l', 'd', 'o', 'o', 'r', ' ', 'o', '!', 'd', 'l', 'e'] ,fittness: 360  Mean: 639.5068359375 ,Variance: 279.5068359375 Time :  0.3202336999999993  ticks: 0.320307016372680 66
Best: ['h', 'e', 'h', 'P', 'h', 'r', ' ', 'o', 'e', 'a', 'd', '!'] ,fittness: 330  Mean: 554.8388671875 ,Variance: 224.8388671875 Time :  0.3886030000000023  ticks: 0.388271570205688 5
Best: [' ', '!', 'l', 'A', 'o', ' ', ' ', 'o', '!', 'd', 'l', 'e'] ,fittness: 300  Mean: 473.5107421875 ,Variance: 173.5107421875 Time :  0.4672918999999993  ticks: 0.467149734497070 3
Best: [' ', '!', 'l', 'A', 'o', ' ', '!', 'o', 'r', 'e', 'd', '!'] ,fittness: 210  Mean: 397.32421875 ,Variance: 187.32421875 Time :  0.5332635999999997  ticks: 0.533124208450317 4
Best: ['h', 'e', 'h', 'l', 'o', ' ', 'W', 'l', '!', 'l', 'h', '!'] ,fittness: 120  Mean: 330.5712890625 ,Variance: 210.5712890625 Time :  0.6054131999999992  ticks: 0.605118751525878 9
Best: ['h', 'e', 'h', 'l', 'o', ' ', 'W', 'l', '!', 'l', 'h', '!'] ,fittness: 120  Mean: 269.12109375 ,Variance: 149.12109375 Time :  0.6728965000000002  ticks: 0.672626495361328 1
Best: ['h', 'e', 'h', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'h', '!'] ,fittness: 60  Mean: 215.7275390625 ,Variance: 155.7275390625 Time :  0.7239734000000002  ticks: 0.723500013351440 4
Best: ['h', 'e', 'h', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'h', '!'] ,fittness: 60  Mean: 171.796875 ,Variance: 111.796875 Time :  0.7769022999999979  ticks: 0.776342391967773 4
Best: ['h', 'e', '!', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!'] ,fittness: 30  Mean: 132.392578125 ,Variance: 102.392578125 Time :  0.829520699999998  ticks: 0.829449176788330 1
Best: ['h', 'e', '!', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!'] ,fittness: 30  Mean: 106.2158203125 ,Variance: 76.2158203125 Time :  0.8739860000000004  ticks: 0.873579263671133 8
Best: ['h', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!'] ,fittness: 0  Mean: 83.73046875 ,Variance: 83.73046875 Time :  0.9375227000000006  ticks: 0.937360763549804 7
```

Two point crossover, with distance heuristic:

```
Best: ['v', 'p', 'p', 'z', 'p', 'I', '6', 'u', 'V', 'p', 'O', '!'] ,fittness: 177  Mean: 409.02880859375 ,Variance: 232.02880859375 Time :  0.10609499999999983  ticks: 0.1063075065612793
Best: ['M', 'f', 'z', 'f', 'n', '%', '_', 'p', 'x', 'a', 'N', 'H'] ,fittness: 141  Mean: 308.85302734375 ,Variance: 167.85302734375 Time :  0.1547194999999988  ticks: 0.15425467491149902
Best: ['z', 'U', '`', 'v', 'j', '#', 'Y', 'a', 'q', 'l', 'g', '7'] ,fittness: 106  Mean: 242.6953125 ,Variance: 136.6953125 Time :  0.19669500000000006  ticks: 0.19667649269104004
Best: ['c', 'f', 'z', 'f', 'n', '%', '_', 'p', 'x', 'g', '`', '-'] ,fittness: 68  Mean: 192.7783203125 ,Variance: 124.7783203125 Time :  0.23865339999999868  ticks: 0.23840594291687012
Best: ['c', 'f', 'z', 'q', 'n', '%', '_', 'p', 'x', 'g', '`', '-'] ,fittness: 67  Mean: 151.08984375 ,Variance: 84.08984375 Time :  0.2801933999999999  ticks: 0.28030928028686523
Best: ['m', 's', 'j', 'd', 's', ' ', 'V', 'j', 's', 'i', 'j', '$'] ,fittness: 52  Mean: 118.07177734375 ,Variance: 66.07177734375 Time :  0.32383449999999847  ticks: 0.32324695587158203
Best: ['o', 'f', 'j', 'j', 'h', '+', 'P', 'r', 'o', 'l', 'c', '#'] ,fittness: 45  Mean: 92.3856640625 ,Variance: 47.3856640625 Time :  0.3765466999999987  ticks: 0.3767855167388916
Best: ['f', 'd', 'd', 'p', 'm', ' ', 'X', 'm', 'x', 'l', 'g', '%'] ,fittness: 33  Mean: 74.388671875 ,Variance: 41.388671875 Time :  0.4285873999999996  ticks: 0.4285612106323242
Best: ['f', 'd', 'd', 'p', 'm', ' ', 'X', 'm', 'x', 'l', 'g', '%'] ,fittness: 33  Mean: 61.76318359375 ,Variance: 28.76318359375 Time :  0.4824053999999993  ticks: 0.4825901985168457
Best: ['f', 'f', 'p', 'm', 'q', ' ', 'V', 'p', 'u', 'o', 'g', '%'] ,fittness: 25  Mean: 53.24365234375 ,Variance: 28.24365234375 Time :  0.5238041999999989  ticks: 0.5236630439758301
Best: ['f', 'h', 'k', 'n', 'l', '#', 'U', 'o', 'p', 'l', 'd', ' '] ,fittness: 19  Mean: 45.232421875 ,Variance: 26.232421875 Time :  0.5735481  ticks: 0.5733025074005127
Best: ['h', 'c', 'k', 'l', 'p', '!', '\\', 'p', 'q', 'l', 'd', '$'] ,fittness: 15  Mean: 39.279296875 ,Variance: 24.279296875 Time :  0.6369680999999989  ticks: 0.6363730430603027
Best: ['h', 'c', 'k', 'l', 'p', ' ', 'V', 'p', 'r', 'n', 'e', '!'] ,fittness: 9  Mean: 33.8466796875 ,Variance: 24.8466796875 Time :  0.6848846999999996  ticks: 0.6844640015350342
Best: ['h', 'c', 'k', 'l', 'p', ' ', 'V', 'p', 'r', 'l', 'e', '!'] ,fittness: 7  Mean: 28.52498234375 ,Variance: 21.52490234375 Time :  0.7284723  ticks: 0.7284953594207764
Best: ['h', 'c', 'k', 'l', 'p', ' ', 'V', 'p', 'r', 'l', 'e', '!'] ,fittness: 7  Mean: 24.529296875 ,Variance: 17.529296875 Time :  0.7686306999999992  ticks: 0.768111467361450
Best: ['h', 'd', 'k', 'l', 'o', ' ', 'V', 'p', 'q', 'l', 'd', '!'] ,fittness: 5  Mean: 20.0009765625 ,Variance: 15.0009765625 Time :  0.8223132  ticks: 0.8214811192321777
Best: ['h', 'e', 'k', 'l', 'p', ' ', 'V', 'o', 's', 'l', 'd', '!'] ,fittness: 4  Mean: 16.33251953125 ,Variance: 12.33251953125 Time :  0.9037378  ticks: 0.9033193588256836
Best: ['h', 'e', 'k', 'l', 'p', ' ', 'V', 'o', 's', 'l', 'd', '!'] ,fittness: 4  Mean: 15.35888671875 ,Variance: 11.35888671875 Time :  0.9547356999999987  ticks: 0.9543159008026123
Best: ['h', 'e', 'k', 'l', 'o', ' ', 'V', 'o', 'r', 'l', 'd', '!'] ,fittness: 2  Mean: 14.03759765625 ,Variance: 12.03759765625 Time :  1.0134394999999987  ticks: 1.0133326053619385
Best: ['h', 'e', 'k', 'l', 'o', ' ', 'V', 'o', 'r', 'l', 'd', '!'] ,fittness: 2  Mean: 12.294921875 ,Variance: 10.294921875 Time :  1.0590772999999984  ticks: 1.058558464050293
Best: ['h', 'e', 'k', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!'] ,fittness: 1  Mean: 10.8857421875 ,Variance: 9.8857421875 Time :  1.1201961999999988  ticks: 1.119968448181152
Best: ['h', 'e', 'k', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!'] ,fittness: 1  Mean: 10.08203125 ,Variance: 9.08203125 Time :  1.1613088999999999  ticks: 1.1614444255828857
Best: ['h', 'e', 'k', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!'] ,fittness: 1  Mean: 9.35546875 ,Variance: 8.35546875 Time :  1.22513  ticks: 1.225258244140625
Best: ['h', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!'] ,fittness: 0  Mean: 8.60089765625 ,Variance: 8.60089765625 Time :  1.2709281999999984  ticks: 1.2705450057983398
```

Two point crossover, with Bulls and Cows heuristic:

```
Best: ['d', 'A', 'l', 'l', ':', 'u', ' ', 'z', '_', 'N', 'd', 'P'] ,fittness: 690  Mean: 1004.0185546875 ,Variance: 314.0185546875 Time :  0.13047650000000033  ticks: 0.13021636009216309
Best: ['d', 'A', 'l', 'l', ':', 'u', ' ', 'z', '_', 'N', 'd', 'P'] ,fittness: 690  Mean: 910.5615234375 ,Variance: 220.5615234375 Time :  0.17597680000000082  ticks: 0.17546772956848145
Best: ['t', 'V', '!', 'G', 'e', 'd', 'N', 'a', 'd', 'W', 'r', '!'] ,fittness: 630  Mean: 824.35546875 ,Variance: 194.35546875 Time :  0.2349172000000017  ticks: 0.23459219932556152
Best: ['h', '0', 'r', '"', 'l', 'l', 'W', 'd', 'l', 'l', 'o', 'E'] ,fittness: 450  Mean: 738.3837890625 ,Variance: 288.3837890625 Time :  0.28072039999999987  ticks: 0.27997565269470215
Best: ['h', 'e', 'N', 'l', '+', 'h', 'W', 'r', 'l', 'l', 'o', 'E'] ,fittness: 390  Mean: 654.8583984375 ,Variance: 264.8583984375 Time :  0.3338686000000024  ticks: 0.33393073081970215
Best: ['!', 'e', 'W', 'h', '!', 'h', 'l', 'd', 'h', 'l', 'l', '!'] ,fittness: 270  Mean: 567.83203125 ,Variance: 297.83203125 Time :  0.39836660000000028  ticks: 0.39841675758361816
Best: ['h', 'e', 'l', 'l', 'l', '(', 'o', 'o', '!', 'l', 'd', 'r'] ,fittness: 210  Mean: 478.974609375 ,Variance: 268.974609375 Time :  0.46745340000000013  ticks: 0.4666910171508789
Best: ['h', 'e', 'W', 'o', '!', ' ', 'e', 'd', 'r', 'l', 'd', 'r'] ,fittness: 180  Mean: 399.66796875 ,Variance: 219.66796875 Time :  0.5206384000000028  ticks: 0.5201699733734131
Best: ['h', 'e', 'l', 'l', 'l', 'e', 'W', 'o', '!', 'l', 'd', 'r'] ,fittness: 120  Mean: 325.4150390625 ,Variance: 205.4150390625 Time :  0.5892271999999998  ticks: 0.5893268585205078
Best: ['h', 'e', 'l', 'l', 'l', ' ', 'W', 'o', 'l', 'l', 'd', 'h'] ,fittness: 90  Mean: 260.91796875 ,Variance: 170.91796875 Time :  0.6669324000000003  ticks: 0.6666343212127686
Best: ['h', 'e', 'h', 'l', 'W', ' ', 'W', 'o', 'r', 'l', 'd', '!'] ,fittness: 60  Mean: 203.6865234375 ,Variance: 143.6865234375 Time :  0.7186538000000002  ticks: 0.7186059951782227
Best: ['h', 'e', 'l', 'l', 'o', ' ', 'W', 'd', 'r', 'l', 'd', '!'] ,fittness: 30  Mean: 157.412109375 ,Variance: 127.412109375 Time :  0.7575801999999996  ticks: 0.7575030326843262
Best: ['h', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!'] ,fittness: 0  Mean: 121.611328125 ,Variance: 121.611328125 Time :  0.8028314999999999  ticks: 0.8025274276733398
```

Uniform crossover, with distance heuristic:

```
Best: ['m', 'X', 'n', 'W', 'l', '0', '`', '4', 'r', 'M', 'p', '!'] ,fittness: 171  Mean: 411.3740234375 ,Variance: 240.3740234375 Time :  0.13301350000000056  ticks: 0.1337597370147705
Best: ['Y', 'J', 'j', '3', 'r', '!', 'R', 'l', 'g', 'n', 'c', '%'] ,fittness: 131  Mean: 301.978515625 ,Variance: 170.978515625 Time :  0.20031350000000003  ticks: 0.20105719566345215
Best: ['b', 'n', 'e', 'j', 'm', '-', 'K', 'O', 'q', '\\', 'g', ')'] ,fittness: 111  Mean: 227.60546875 ,Variance: 116.60546875 Time :  0.26536639999999956  ticks: 0.2660183906555176
Best: ['k', 'm', 'd', 'z', 'n', '"', 'W', 'u', 'i', 'r', '`', ' '] ,fittness: 67  Mean: 170.81103515625 ,Variance: 103.81103515625 Time :  0.32807099999999956  ticks: 0.3286197185516357
Best: ['r', 'p', 'e', 'm', 's', '#', 'W', 'q', 'r', 'n', 'b', '#'] ,fittness: 44  Mean: 128.64013671875 ,Variance: 84.64013671875 Time :  0.3962560000000105  ticks: 0.3961644172668457
Best: ['b', 'e', 'g', 'n', 'm', ' ', 'U', 'q', 'u', 'p', '\\', '"'] ,fittness: 40  Mean: 97.9306640625 ,Variance: 57.9306640625 Time :  0.47076580000000057  ticks: 0.4711306095123291
Best: ['k', '^', 'j', 'g', 'l', ' ', 'V', 'p', 's', 'r', 'e', ' '] ,fittness: 31  Mean: 75.591796875 ,Variance: 44.591796875 Time :  0.5351941999999994  ticks: 0.5352756977081299
Best: ['g', 'g', 'n', 'k', 'w', '#', 'Y', 'n', 's', 'l', 'd', '!'] ,fittness: 21  Mean: 58.39794921875 ,Variance: 37.39794921875 Time :  0.5838303000000007  ticks: 0.5841684341430664
Best: ['h', 'l', 'l', 'k', 's', ' ', 'W', 'q', 's', 'l', 'd', '!'] ,fittness: 15  Mean: 46.61181640625 ,Variance: 31.61181640625 Time :  0.6510820000000006  ticks: 0.6513195037841797
Best: ['h', 'e', 'l', 'k', 'm', ' ', 'X', 'o', 's', 'n', 'e', '#'] ,fittness: 10  Mean: 35.52197265625 ,Variance: 25.52197265625 Time :  0.7312989999999999  ticks: 0.7316086292266846
Best: ['h', 'e', 'l', 'l', 'l', ' ', 'V', 'p', 's', 'l', 'e', '!'] ,fittness: 7  Mean: 29.5625 ,Variance: 22.5625 Time :  0.7946454999999997  ticks: 0.7954692840576172
Best: ['h', 'e', 'l', 'k', 'p', ' ', 'V', 'm', 'r', 'k', 'd', '!'] ,fittness: 6  Mean: 23.859375 ,Variance: 17.859375 Time :  0.8451690000000003  ticks: 0.8452482223510742
Best: ['h', 'd', 'l', 'l', 'n', ' ', 'X', 'n', 'r', 'l', 'c', '!'] ,fittness: 5  Mean: 20.1162109375 ,Variance: 15.1162109375 Time :  0.9249840999999996  ticks: 0.9256322383880615
Best: ['h', 'd', 'l', 'l', 'p', ' ', 'W', 'p', 'r', 'l', 'd', '!'] ,fittness: 3  Mean: 16.619140625 ,Variance: 13.619140625 Time :  1.0152847999999999  ticks: 1.0156364440917969
Best: ['h', 'e', 'l', 'k', 'p', ' ', 'W', 'o', 'r', 'l', 'd', '!'] ,fittness: 2  Mean: 13.37060546875 ,Variance: 11.37060546875 Time :  1.0921806000000007  ticks: 1.0923051834106445
Best: ['h', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!'] ,fittness: 0  Mean: 12.68994140625 ,Variance: 12.68994140625 Time :  1.2234411000000005  ticks: 1.224266767501831
```

Uniform crossover, with Bulls and Cows heuristic:

```
Best: ['H', 'e', '\\', '\\', 'l', 'd', 'b', 'h', '!', '@', 'W', ';'] ,fittness: 690  Mean: 1005.1904296875 ,Variance: 315.1904296875 Time :  0.13821949999999994  ticks: 0.13802647590637207
Best: ['H', 'e', '\\', '\\', 'l', 'd', 'b', 'h', '!', '@', 'W', ';'] ,fittness: 690  Mean: 907.7783203125 ,Variance: 217.7783203125 Time :  0.1874728000000001  ticks: 0.1871647834777832
Best: ['H', 'e', 'e', 'l', 'l', 'd', 'c', 'h', '!', 'h', 'J', ';'] ,fittness: 540  Mean: 809.0771484375 ,Variance: 269.0771484375 Time :  0.2355245000000036  ticks: 0.23492145538330078
Best: ['d', 'd', 'r', 'l', '#', 'e', '!', 'r', 'r', ' ', 'd', '#'] ,fittness: 390  Mean: 706.34765625 ,Variance: 316.34765625 Time :  0.28585559999999965  ticks: 0.28543734550476074
Best: ['t', 'W', 'h', 'r', 'h', '!', 'W', 'o', 'r', 'h', 'd', 'Q'] ,fittness: 360  Mean: 600.439453125 ,Variance: 240.439453125 Time :  0.3437629999999915  ticks: 0.3432631492614746
Best: ['o', 'e', 'l', 'l', 'd', '+', 'W', 'o', 'r', 'd', 'e', 'h'] ,fittness: 240  Mean: 493.0078125 ,Variance: 253.0078125 Time :  0.4070552999999997  ticks: 0.4066042900085449
Best: ['h', 'r', '!', 'l', 'o', ' ', 'W', '[', 'r', 'l', 'h', '!'] ,fittness: 180  Mean: 392.4609375 ,Variance: 212.4609375 Time :  0.4641809999999989  ticks: 0.4636094570159912
Best: ['h', 'e', 'l', 'o', 'l', 'l', 'W', 'o', 'r', ' ', 'd', '!'] ,fittness: 120  Mean: 306.6650390625 ,Variance: 186.6650390625 Time :  0.5208610999999994  ticks: 0.5203602313995361
Best: ['h', 'e', 'l', 'l', 'e', 'e', 'W', 'o', 'r', '!', 'd', '!'] ,fittness: 90  Mean: 233.1005859375 ,Variance: 143.1005859375 Time :  0.5854578999999998  ticks: 0.5853838920593262
Best: ['h', 'e', 'l', 'l', 'o', 'l', 'W', 'o', 'r', 'l', 'd', 'o'] ,fittness: 60  Mean: 177.01171875 ,Variance: 117.01171875 Time :  0.6489593999999999  ticks: 0.6485602855682373
Best: ['h', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!'] ,fittness: 0  Mean: 131.0888671875 ,Variance: 131.0888671875 Time :  0.7098383999999989  ticks: 0.7090597152709961
```

.7

החלק שבו אנחנו עושים EXPLORATION זה כאשר עושים crossover or mutate כי אנו מחפשים את הילדים החדשים שנוצרו.

החלק שבו אנחנו עושים EXPLOITATION זה כאשר מעבירים את אלה ששרדו לדור הבא (elitism) לפי ערך ה fitness, כי בו אנחנו חוזרים על מצבים שכבר מוכרים לנו.

.8

הסבר כללי :

יצרנו class שמטפל בכל סוג של בעיה שבו יש :

1. פונקציית יצור האובייקט create_object
2. פונקציית יצור איבר באובייקט character_creation

```
class parameters:
    fitnesstype = fitness_selector().select

    def __init__(self):
        self.object = None
        self.fitness = 0

    # creates a member of the population
    def create_object(self, target_size, target):
        return self.object

    def character_creation(self, target_size):
        pass

    # function to calculate the fitness for this specific problem
    def calculate_fittness(self, target, target_size, select_fitness, age_update=True):
        # print(len(self.object))
        self.fitness = self.fitnesstype[select_fitness](self.object, target, target_size)
        return self.fitness

    def helper(self, target_size):
        pass

    # for sorting purposes
    def __lt__(self, other):
        return self.fitness < other.fitness
```

משתמשים בclass parameters כבסיס לכל variation של הבעיה

למשל עבור בול פגיעה :

```python
# class for first problem
class DNA(parameters):
    mutation = mutations()

    def __init__(self):
        parameters.__init__(self)

    def create_object(self, target_size, target):
        self.object = []
        for j in range(target_size):
            self.object += [self.character_creation(target_size)]
        self.helper(target_size)
        return self.object

    def character_creation(self, target_size):
        return chr((random.randint(0, 90)) + 32)

    def mutate(self, target_size, member, mutation_type):
        self.mutation.select[mutation_type](target_size, member, self.character_creation)
```

לאלגוריתם PSO השתמשנו באותו מבנה של אובייקט של בול פגיעה שקראנו לו DNA והוספנו 3 פרמטרים כדי להתאים את בול פגיעה לבעיית PSO :

1. velocity.
2. p_best : מכיל את הפיטניס הכי טוב של ה particle הנוכחי.
3. p_best_object : מכיל את האובייקט הכי טוב שהיה ל particle הזה.

פונקציות :

1. calculate_velocity שמחשב את ה velocity לפי ה gl_best שהוא global minima שמחשב את ה velocity לפי ה gl_best שהוא global minima

```python
class PSO_prb(DNA):
    # our object is the initial position , we added 2 parameters that are required
    def __init__(self):
        DNA.__init__(self)
        self.velocity = None
        self.p_best = sys.maxsize
        self.p_best_object = None

    def helper(self, target_size):
        self.create_velocity(target_size)

    def create_velocity(self, target_size):
        self.velocity = [random.random() for i in range(target_size)]

    def calculate_new_position(self):
        pos = ""
        for i in range(len(self.object)):
            pos += chr((ord(self.object[i]) + int(self.velocity[i])) % 256)
        self.object = pos

    def calculate_velocity(self, c1, c2, gl_best, w=0.5):
        for i in range(len(self.object)):
            cc1 = c1 * (ord(self.p_best_object[i]) - ord(self.object[i])) * random.random()
            cc2 = c2 * (ord(gl_best[i]) - ord(self.object[i])) * random.random()
            self.velocity[i] = self.velocity[i] * w + cc1 + cc2

    def __eq__(self, other):
        DNA.__eq__(self, other)
        self.velocity = other.velocity
        self.best_object = other.best_object
```

אלגוריתם :

```python
class PSO_alg(algortithem):
    def __init__(self, target, tar_size, pop_size, problem_spec, fitnesstype, selection=None):
        algortithem.__init__(self, target, tar_size, pop_size, problem_spec, fitnesstype, selection)
        self.global_minima = problem_spec()
        self.local_minima = problem_spec()

    def calc_fitness(self):...

    def algo(self, i):
        if i == 0:
            self.global_minima.object = self.population[0].object
            self.global_minima.calculate_fittness(self.target, self.target_size, self.fitnesstype)
        self.calc_fitness()
        self.sort_by_fitness()

        w = ((i - GA_MAXITER) * 0.4) / (GA_MAXITER ** 2 + 0.4)
        c1 = ((-3 * i) / GA_MAXITER) + 2
        c2 = ((3 * i) / GA_MAXITER) + 2
        if not self.stopage():
            for particle in self.population:
                particle.calculate_velocity(c1, c2, self.global_minima.object, w)
                particle.calculate_new_position()

    def stopage(self):
        return self.global_minima.fitness == 0
```

.9

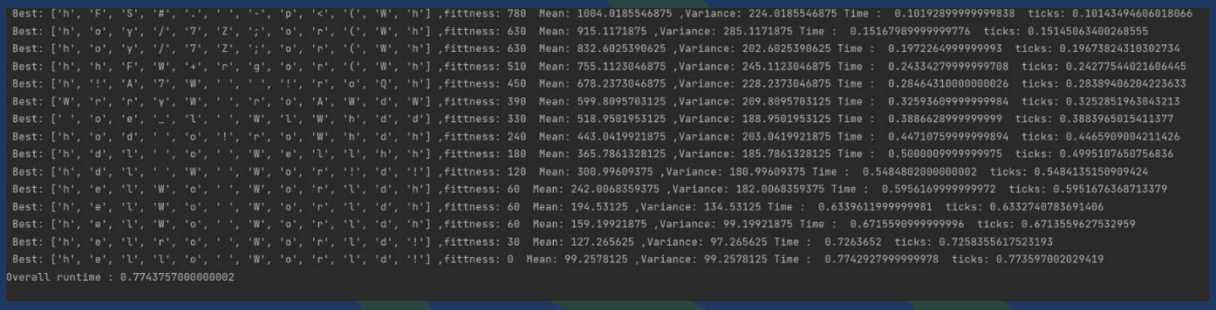
1. כאשר האוכלוסייה גדולה הביצועים של PSO יותר טובים מבחינת זמן ריצה ואיכות תשובה, היא מחזירה תשובה בפחות זמן ובפחות מספר של דורות.

2. כאשר האוכלוסייה קטנה אין הבדל משמעותי בין PSO ל genetic algorithm

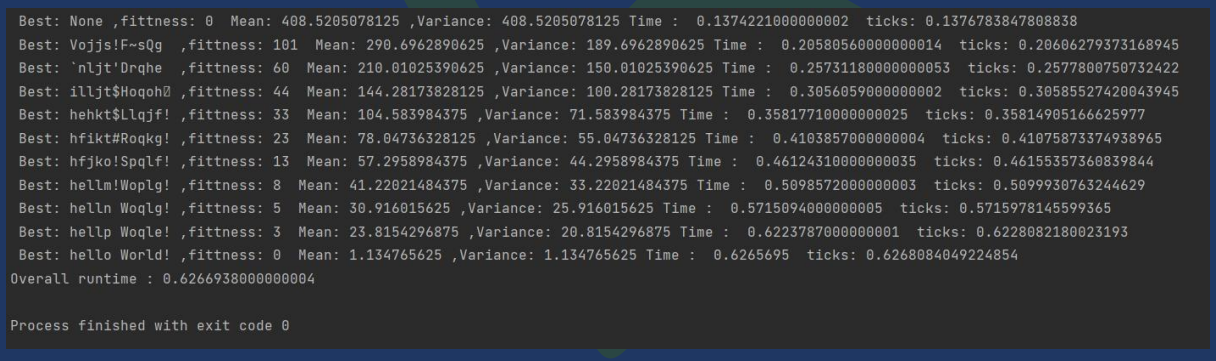
פלט:

*אוכלוסייה גדולה: 2048

Genetic:

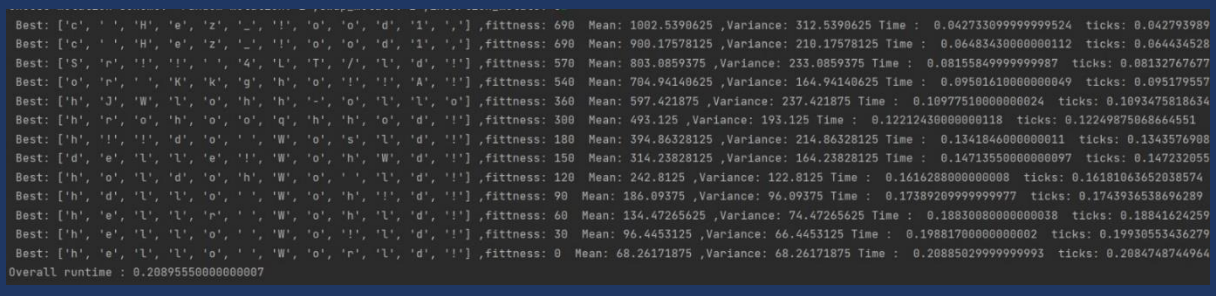


PSO:



*אוכלוסייה קטנה: 512

Genetic:



PSO:

```
Best: None ,fittness: 0  Mean: 405.697265625 ,Variance: 405.697265625 Time :  0.033126199999999884  ticks: 0.03269672393798828
Best: lntJ⍰$U\⍰pZ) ,fittness: 137  Mean: 307.599609375 ,Variance: 170.599609375 Time :  0.05172440000000034  ticks: 0.0520319938659668
Best: aksg^!Simo\% ,fittness: 73  Mean: 243.890625 ,Variance: 170.890625 Time :  0.07487099999999991  ticks: 0.07446694374084473
Best: ehkke!Omvq[% ,fittness: 51  Mean: 192.23046875 ,Variance: 141.23046875 Time :  0.09368100000000013  ticks: 0.09353494644165039
Best: kjmlo⍰Rerh`# ,fittness: 36  Mean: 141.677734375 ,Variance: 105.677734375 Time :  0.10432039999999976  ticks: 0.10450243949890137
Best: djlkq⍰Vnria# ,fittness: 25  Mean: 111.380859375 ,Variance: 86.380859375 Time :  0.11851520000000004  ticks: 0.1184840202331543
Best: hjkkp Tlrma# ,fittness: 20  Mean: 86.81640625 ,Variance: 66.81640625 Time :  0.1340028999999996  ticks: 0.13347220420837402
Best: hilkp⍰Wnrma# ,fittness: 15  Mean: 69.935546875 ,Variance: 54.935546875 Time :  0.1454810000000002  ticks: 0.14578700065612793
Best: hikkp⍰Wnrmd! ,fittness: 11  Mean: 56.474609375 ,Variance: 45.474609375 Time :  0.1561287  ticks: 0.15643811225891113
Best: hillp Worme" ,fittness: 8  Mean: 45.421875 ,Variance: 37.421875 Time :  0.17366360000000025  ticks: 0.1733853816986084
Best: hhllp Xormd" ,fittness: 7  Mean: 38.216796875 ,Variance: 31.216796875 Time :  0.18303660000000033  ticks: 0.18335866928100586
Best: hgllp Wormd! ,fittness: 4  Mean: 34.783203125 ,Variance: 30.783203125 Time :  0.19237069999999967  ticks: 0.19184637069702148
Best: hellp Wormd! ,fittness: 2  Mean: 30.13671875 ,Variance: 28.13671875 Time :  0.20106290000000016  ticks: 0.20145416259765625
Best: hello Wormd! ,fittness: 1  Mean: 25.31640625 ,Variance: 24.31640625 Time :  0.2145849000000002  ticks: 0.21493911743164062
Best: hello World! ,fittness: 0  Mean: 0.00390625 ,Variance: 0.00390625 Time :  0.2148772000000001  ticks: 0.21493911743164062
Overall runtime : 0.21490489999999962
```

חלק ב

.1

spin the roulette is used in main if rws is used

```
def SUS(self, population, fitness_array, k=10):
    # get comulative sum of all fitness values
    fitness_comulative = fitness_array.cumsum()
    # wheel steps each time we choose
    steps = fitness_comulative[-1] / 2
    # select a random place to begin between 0 and our steps
    begin = random() * steps
    # here we get two evenly spaced places in wheel  !
    new_selection = numpy.arange(begin, fitness_comulative[-1], steps)
    [i1, i2] = numpy.searchsorted(fitness_comulative, new_selection)
    return population[i1], population[i2]
```

```
def RWS(self, population, fitness_array, k=10):
    # check the +1 !
    range_of_choices = len(self.ranks)
    # roll the rullette
    chosen = numpy.random.choice(range_of_choices, p=self.ranks)
    chosen2 = numpy.random.choice(range_of_choices, p=self.ranks)
    return population[chosen], population[chosen2]
```

```
def tournement(self, population, fitness_array, k=15):
    # get samples from population
    participants1 = sample(population, k)
    participants2 = sample(population, k)
    # return minumum from 2 samples
    return min(participants1),min(participants2)
```

```
def spin_the_rulette(self, population, mean):
    # spin the wheel:
    fitness_array = numpy.array([1 / linear_scale((citizen.fitness + 1, 0.5, 0)) for citizen in population])
    fitness_sum = fitness_array.sum()
    self.ranks = [1 / linear_scale((citizen.fitness + 1, 0.5, 0)) / fitness_sum for citizen in population]
```

.2

הוספנו פרמטר חדש לכל אובייקט age:

```
class parameters:
    fitnesstype = fitness_selector().select

    def __init__(self):
        self.object = None
        self.age = 0
        self.fitness = 0
```

מעדכנים אותו בכל חישוב של fitness (בכל דור פעם אחת):

```
def calculate_fittness(self, target, target_size, select_fitness, age_update=True):
    # print(len(self.object))
    self.fitness = self.fitnesstype[select_fitness](self.object, target, target_size)
    self.age += 1 if age_update else 0
    return self.fitness
```

בפונקציית mate באלגוריתם הגנטי השתמשנו בפונקציית age_based שמחזירה את האוכלוסייה ששורדת לדור הבא:

```
def age_based(self):
    age_based_population = [citizen for citizen in self.population if 2 < citizen.age < 20]
    self.buffer[:len(age_based_population)] = age_based_population[:]
    return len(age_based_population)
```

.3

```
# class to define n queens problem
# approach :  with an array of N places , each place represents the row
# and the value in each place represents colums meaning :
# Arr={6,3,...}   ;   Arr[0] is the 6's column and 0 is the row
class NQueens_prb(DNA):
    def __init__(self):
        parameters.__init__(self)

    def create_object(self, target_size, target):
        obj = random.sample(range(target_size), target_size)
        while len(unique(obj)) != len(obj):
            obj = random.sample(range(target_size), target_size)
        self.object = obj

    def character_creation(self, target_size):
        return random.randint(0, target_size - 1)
```

.4

ממשנו 2 מוטציות חדשות , swap ו insertion:

```
class mutations:
    def __init__(self):
        self.select = {1: self.random_mutate, 2: self.swap_mutate, 3: self.insertion_mutate,BIN:self.distroy_mutate}

    def random_mutate(self, target_size, member, character_creation):
        ipos = random.randint(0, target_size - 1)
        delta = character_creation(target_size)
        member.object = member.object[:ipos] + [delta] + member.object[ipos + 1:]

    def swap_mutate(self, target_size, member, character_creation):
        ipos = random.randint(0, target_size - 2)
        ipos2 = random.randint(ipos + 1, target_size - 1)
        member.object = member.object[:ipos] + [member.object[ipos2]] + member.object[ipos + 1:ipos2] + [
            member.object[ipos]] + member.object[ipos2 + 1:]

    def insertion_mutate(self, target_size, member, character_creation):
        ipos = random.randint(0, target_size - 2)
        ipos2 = random.randint(ipos + 1, target_size - 1)
        member.object = member.object[:ipos] + member.object[ipos + 1:ipos2] + [member.object[ipos]] + member.object[
                                                                                                       ipos2:]
```

מימשנו את PMX ו CX ב Class cross_types:

```python
def PMX(self, citizen1, citizen2):
    target_size =min (len(citizen1),len(citizen2))
    # repeat 5 times
    for j in range(5):
        ipos = random.randint(0, target_size - 1)
        c1 = citizen1.object[ipos]
        c2 = citizen2.object[ipos]
        # first mutation
        object1 = citizen1.object[0:ipos] + [c2] + citizen1.object[ipos + 1:]
        object2 = citizen2.object[0:ipos] + [c1] + citizen2.object[ipos + 1:]

        for i in range(target_size):
            object1[i] = c2 if object1[i] == c1 else c1 if object1[i] == c2 else object1[i]
            object2[i] = c1 if object2[i] == c2 else c2 if object2[i] == c1 else object2[i]

    return object1, object2
```

```python
# problem accures when cyrcle is broken !
def CX(self, citizen1, citizen2):
    object1 = citizen1.object
    object2 = citizen2.object
    target_size = min(len(citizen1.object), len(citizen2.object))
    hash1 = {citizen1.object[i]: i for i in range(target_len)}
    cycles = []
    cycle = []
    all_indexes = []
    # get all cycles
    if len(unique(object1)) - len(object1) != len(unique(object2)) - len(object2):
        print(f"problem doesn't support this type of crossing ! charachters need to be unique ! so that the cycles exist ")
    # find the cycles
    for i in range(target_len):
        # if we havent gone over this cycle then get its members
        if i not in all_indexes:
            self.cycle(hash1, i, citizen1.object,citizen2.object, cycle)
            cycles.append(cycle)
            all_indexes = all_indexes[:] + cycle[:]
    for i in range(len(cycles)):
        # if current cycle devides by 2 then swap (i+1 because cycles->(1,...n))
        if (i+1) % 2 == 1:
            current_cycle = cycles[i]
            for j in current_cycle:
            # swap the values in this specific cycle
                object1[j], object2[j] = object2[j], object1[j]
    return object1, object2
```

.5

קריטריונים	בול פגיעה	N **מלכות**
גודל האוכלוסייה	אם מגדילים גודל האוכלוסייה מספר הדורות קטן אבל זמן ריצה יותר גדול	אם מגדילים גודל האוכלוסייה מספר הדורות קטן אזל זמן ריצה יותר גדול
הסתברות למוטציות	אם מגדילים את הסתברות מוטציה רואים שמספר דורות וזמן ריצה קטן. אבל לפעמים הוא גדל בהפרשים קטנים, כלומר דור אחד ו הפרש של פחות משנייה.	אם מגדילים הסתברות מוטציות אז מספר הדורות וזמן הרצה קטן.
אסטרטגיית הבחירה	אסטרטגיית הטורניר הייתה הכי מהירה	אסטרטגיית הטורניר הייתה הכי מהירה
אסטרטגית השרידות	שרידות לפי ה AGING הכי טובה	שרידות לפי ה AGING הכי טובה
אסטרטגיית שיחלוף	ה uniform cross היה הכי טוב	ה CX היה הכי טוב

.6

אלגוריתם MINIMAL CONFLICTS נתן ביצועים מאוד מהירים ביחס ל- genetic algorithm, תמיד החזיר תשובה נכונה.

MINIMAL CONFLICTS:

```
Best: [3, 8, 3, 9, 0, 2, 5, 1, 6, 4] ,fittness: 2  Mean: 0 ,Variance: 2.0 Time :  0.003930400000000223  ticks: 0.0035271644592285156
Best: [3, 8, 4, 9, 0, 2, 5, 1, 6, 4] ,fittness: 2  Mean: 0 ,Variance: 2.0 Time :  0.0040181000000001355  ticks: 0.0035271644592285156
Best: [3, 8, 4, 9, 0, 2, 5, 1, 6, 1] ,fittness: 2  Mean: 0 ,Variance: 2.0 Time :  0.00413730000000001  ticks: 0.0035271644592285156
Best: [3, 8, 4, 9, 0, 2, 5, 1, 6, 4] ,fittness: 2  Mean: 0 ,Variance: 2.0 Time :  0.004207500000000142  ticks: 0.0035271644592285156
Best: [3, 8, 3, 9, 0, 2, 5, 1, 6, 4] ,fittness: 2  Mean: 0 ,Variance: 2.0 Time :  0.00427549999999877  ticks: 0.004499912261962891
Best: [3, 8, 3, 9, 0, 2, 5, 1, 6, 4] ,fittness: 2  Mean: 0 ,Variance: 2.0 Time :  0.004410000000000025  ticks: 0.004499912261962891
Best: [3, 8, 7, 9, 0, 2, 5, 1, 6, 4] ,fittness: 2  Mean: 0 ,Variance: 2.0 Time :  0.004497100000000032  ticks: 0.004499912261962891
Best: [3, 8, 7, 9, 0, 2, 5, 1, 6, 4] ,fittness: 2  Mean: 0 ,Variance: 2.0 Time :  0.004565200000000047  ticks: 0.004499912261962891
Best: [3, 1, 7, 9, 0, 2, 5, 1, 6, 4] ,fittness: 2  Mean: 0 ,Variance: 2.0 Time :  0.004676700000000089  ticks: 0.004499912261962891
Best: [3, 1, 7, 9, 0, 2, 5, 1, 6, 4] ,fittness: 2  Mean: 0 ,Variance: 2.0 Time :  0.004744599999999988  ticks: 0.004499912261962891
Best: [3, 1, 7, 9, 0, 2, 5, 8, 6, 4] ,fittness: 2  Mean: 0 ,Variance: 2.0 Time :  0.004814800000000119  ticks: 0.004499912261962891
Best: [3, 1, 7, 9, 0, 2, 5, 8, 6, 4] ,fittness: 0  Mean: 0 ,Variance: 0.0 Time :  0.004861400000000238  ticks: 0.004499912261962891
Overall runtime : 0.004902200000000079
```

Genetic:

```
Best: [1, 3, 9, 7, 8, 5, 0, 2, 4, 6] ,fittness: 2  Mean: 12.6015625 ,Variance: 10.6015625 Time :  0.15939449999999944  ticks: 0.15928006172180176
Best: [6, 9, 2, 4, 0, 3, 7, 5, 1, 8] ,fittness: 2  Mean: 7.07421875 ,Variance: 5.07421875 Time :  0.28159390000000073  ticks: 0.28119850158691406
Best: [8, 3, 1, 7, 2, 6, 4, 0, 5, 9] ,fittness: 2  Mean: 5.6953125 ,Variance: 3.6953125 Time :  0.36550560000000054  ticks: 0.36554861068725586
Best: [4, 5, 0, 6, 9, 7, 1, 3, 8, 2] ,fittness: 2  Mean: 4.2421875 ,Variance: 2.2421875 Time :  0.4750525999999997  ticks: 0.47500061988830566
Best: [4, 5, 0, 6, 9, 7, 1, 3, 8, 2] ,fittness: 0  Mean: 3.0625 ,Variance: 3.0625 Time :  0.6332961000000008  ticks: 0.632716178894043
Overall runtime : 0.6333514000000005
```

.7

First Fit:

Takes our standard class for algorithms and uses our first fit class to fill the bins:

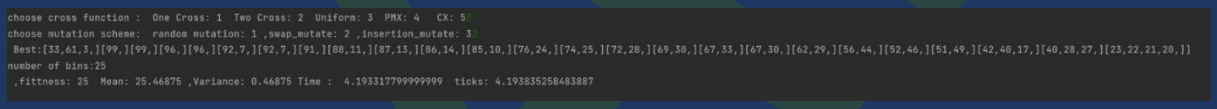
```python
class FirstFit(algortithem):
    def __init__(self, target, tar_size, problem_spec=None, fitness=None, selection=None):
        super(FirstFit, self).__init__(target, tar_size, 1, problem_spec, BIN, selection)
    def algo(self,i):
        bins=self.prob_spec()
        bins.set_capacity(self.target[1])
        bins.target_creater(self.target)
        bins.create_object(self.target_size,self.target)
        bins.calculate_fittness(self.target, self.target_size, self.fitnesstype)
        self.solution=self.population[0]=bins
    def stopage(self):
        return True
```

The self.prob_spec is our first fit algorithm taken from bin class:

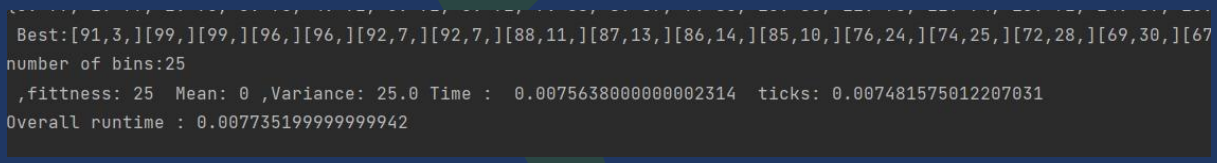
```
class first_fit_prob(bin):
    hash = hash_table
    capacity = 0

    def __init__(self, capacity, items=[], fill=0):
        bin.__init__(self,capacity)

    def fill_bins(self, items):
        for item in range(len(items)):
            if self.fill + self.hash[items[item]] <= self.capacity:
                self.items.append(items[item])
                self.fill += self.hash[items[item]]
        return setdiff1d(items, self.items)
```

Results:

Genetic algorithm:



First Fitness:



Bin packing problem class that we sent to genetic algorithm.

We added bin objects so that we can configure them when calculating fitness, i.e creating chromosome with bins.

```
class bin_packing_prob(DNA):
    target = []
    capacity = 0
    bin1=bin

    def __init__(self):
        parameters.__init__(self)
        self.bin_objects = []

    def target_creater(self, target):
        self.target = target

    def set_capacity(self, cap):
        self.capacity = cap
        self.bin1.capacity = cap

    def create_special_parameter(self, target_size):
        self.bin_objects = []
        obj = self.object
        # print(self)
        while len(obj):
            new_bin = self.bin1(self.capacity)
            obj = new_bin.fill_bins(obj)
            self.bin_objects[:] = self.bin_objects[:] + [new_bin]
        # print(self)

    def create_object(self, target_size, target):
        # print(len(self.target[0]),self.capacity)
        self.object = random.sample(range(len(self.target[0])), len(self.target[0]))
        # self.bin_cappacity = self.target[0]
        self.create_special_parameter(target_size)
```