**חלק א':**

1. כדרוש, נוספו פונקציות לחישוב ממוצע וסטיית תקן של ה-fitness (קובץ GeneticAlgorithm.py שורות 230-238):

# calculates average fitness of current generation  
def avg\_fit(gen\_arr):  
 fit\_arr = [g.fitness for g in gen\_arr]  
 return np.mean(fit\_arr)  
  
  
# calculates STD of current generation  
def std\_fit(gen\_arr):  
 fit\_arr = [g.fitness for g in gen\_arr]  
 return np.std(fit\_arr)

1. הוספת חישוב דיווח זמנים לכל דור בכל ריצה:

<פלט של פייתון בגרף>

1. כדרוש, נוספו אופרטורים לשיחלוף SINGLE, TWO, UNIFORM (קובץ GeneticAlgorithm.py שורות 91-112).
2. כדרוש, נוספה היוריסטיקת "בול פגיעה", כאשר יש "פגיעה" מצ'פרת ב-1 (EXACT\_BONUS), שיש "בול" מצ'פרת ב-10 (HIT\_BONUS). ניתן לראות מימוש היוריסטיקה בקובץ GeneticAlgorithm.py שורות 64-74.
3. השוואת היוריסטיקת "בול פגיעה" להיוריסטיקה המקורית:

* ....

1. החלקים האחראים על EXPLORATION באלגוריתם הגנטי הינם:

* פונקציית mutate והקבוע GA\_MUTATIONRATE – פונקציה זאת יוצרת קונפיגורציה/גן חדש על ידי שינוי רנדומלי בגן נתון, ובכך מנסה להימנע ממינימום לוקאלי ומגדיל את ה-EXPLORATION.

על כן, ככל שנגדיל את ה- GA\_MUTATIONRATEכך יגדל ה-EXPLORATION גם כן.

* ...

החלקים האחראים על EXPLOITATION באלגוריתם הגנטי הינם:

* פונקציית elitism והקבוע GA\_ELITRATE – פונקציה זאת שואפת ליצור דור על סמך הגנים הטובים ביותר בדור הנוכחי, ועל כן מנצלת את התוצאות הטובות ביותר בדור הנוכחי ובכך בא לידי ביטוי ה-EXPLOITION.

על כן, ככל שנגדיל את ה-GA\_ELITRATE כך יגדל ה-EXPLOITION גם כן.

* בחירת הגנים הזכאים לריבוי נעשת (טרם הוספת אופציות זכאות נוספות) בצורה כזאת שנבחרים רק ממחצית הדור בעלי ה-fitness הגובה ביותר. סוג בחירה כזאת מגדיל את ה-EXPLOITION שכן מנצל את הגנים החזקים ביותר עד כה.

1. כדרוש, מומש אלגוריתם PSO לבעיה זו. הפונקציה בקובץ GenticAlgorithm.py שורות 256-280.

כמו כן, ישנה התייחסות לאורך הקוד למקרה בו האלגוריתם PSO רץ, על כן ניתן להריצו עבור שתי היוריסטיקות.

1. <<נריץ את שני האלגוריתמים עם אוכלוסייה גדולה/קטנה, אולי נשחק גם פרמטרים של הPSO, נגביל איטרציות ל2000, נבדוק ונדפיס גרף המשווה>>

**חלק ב':**

1. כדרוש, נוספו שיטות בחירה הבאות:

RWS + scaling – קובץ GenticAlgorithm.py שורות 122-133

SUS + scaling - קובץ GenticAlgorithm.py שורות 136-149

TOURNAMENT - קובץ GenticAlgorithm.py שורות 152-158

כמו כן, שיטת הבחירה הרגילה (מחצית האוכלוסייה הטובה ביותר) - קובץ GenticAlgorithm.py שורות 161-163

פונקציית ה-scaling המשמשת את RWS, SUS - .... קובץ GenticAlgorithm.py שורות 115-119

1. כדרוש, נוספה שיטת שרידות Aging - קובץ GenticAlgorithm.py שורות 195-200:

def ageing(gen\_arr, min\_age):  
 can\_mate = []  
 for g in gen\_arr:  
 if g.age >= min\_age:  
 can\_mate.append(g)  
 return can\_mate

1. כדרוש, נוספה תמיכה לבעיית N המלכות, קובץ NQueens.py.

בהתאם לבעיה, הגן אינו מקבל מחרוזת אקראית כפי שהיה בתרגיל המקורי, ועל כן, מומשה פונקציה המאתחלת את האוכלוסייה בפרמוטציות של המספרים 1-N. קובץ NQueens.py שורות 84-90.

def init\_nqueens():  
 pop, buffer = [], []  
 for i in range(NQ\_POPSIZE):  
 ran\_prem = np.random.permutation(range(1, N+1))  
 pop.append(GA.Genetic(ran\_prem))  
 buffer.append(GA.Genetic(ran\_prem))  
 return pop, buffer

בנוסף, אופן חישוב ה-fitness של הגנים מתבצע אחרת, על כן מומשה פונקציה בהתאם.

קובץ NQueens.py שורות 49-60.

class ConflictsTotal:  
 def calc\_fitness(self, boards):  
 for b in boards:  
 self.calc\_personal\_fitness(b)  
  
 def calc\_personal\_fitness(self, board):  
 total\_fit = 0  
 for i in range(N):  
 for j in range(i+1,N):  
 if board.str[i] == board.str[j] or \

board.str[i] == board.str[j] - (j - i) or \

board.str[i] == board.str[j] + (j - i):  
 total\_fit += 1  
 board.fitness = total\_fit

נציין, כי הבעיה נפתרת על ידי המימוש של הבעיה המקורית (כלומר, בקובץ GaneticAlgorithm.py), בנוסף לפונקציות שנדרשנו לממש בסעיפים הבאים.

1. כדרוש, נוספו **2 אופרטורים לשיחלוף**:

* אופרטור PMX, קובץ NQueens.py שורות 12-27:

class PMXCrossover:  
 def crossover(self, perm1, perm2):  
 iter = random.randint(1,N-1)  
 child = [i for i in perm1]  
  
 # do PMX crossover a random number of times  
 for it in range(iter):  
 index = random.randint(0, N-1)  
 val1 = perm1[index]  
 val2 = perm2[index]  
 for i in range(N):  
 if child[i] == val1:  
 child[i] = val2  
 elif child[i] == val2:  
 child[i] = val1  
 return child

* אופרטור OX, קובץ NQueens.py שורות 31-43:

class OXCrossover:  
 def crossover(self, perm1, perm2):  
 child = [i for i in perm1]  
 # val1 will store half of the values randomly  
 val1 = random.sample(range(0, N), N // 2)  
 # val2 will store the remaining values  
 val2 = [i for i in perm2 if i not in val1]  
 v2 = 0  
 for i in range(len(child)):  
 if child[i] not in val1:  
 child[i] = val2[v2]  
 v2 += 1  
 return child

כדרוש, נוספו **2 אופרטורים למוטציות לתמורות** המתאימים לבעיה:

* מוטציית החלפה, קובץ קובץ NQueens.py שורות 66-71:

class SwapMutation:  
 def mutate(self, gen):  
 index = random.sample(range(0, N), 2)  
 temp = gen.str[index[0]]  
 gen.str[index[0]] = gen.str[index[1]]  
 gen.str[index[1]] = temp

* מוטציית ערבול, קובץ NQueens.py שורות 74-78:

class ScrambleMutation:  
 def mutate(self, gen):  
 start = random.randint(0, N)  
 end = random.randint(start, N)  
 gen.str[start:end] = np.random.permutation(gen.str[start:end])

1. <<המון גרפים – פרמטרים שונים – להריץ באיזה לולאה או משהו כזה>>>
2. הפרמטרים האופטימליים הינם:

* בדרך
* בדרך
* בדרך

1. כדרוש, מומש אלגוריתם Minimal Conflicts לבעיית N המלכות. הקובץ minimalConflictsAlgorithm.py.

אלגוריתם זה, מתכנס תוך מספר איטרציות קטן יחסית. אך לרוב נתקע במינימום לוקאלי.

<< נוסיף גרף של גודל הלוח על אחוז סך הקונפליקטים בסיום הריצה בממוצע 4-30N=>>

לצורך הכלאה בין שני האלגוריתמים, נציע 3 שיטות:

* נוסיף להיוריסטיקה של האלגוריתם הגנטי פונקציה "המענישה" לוחות עם מלכות בעלות קונפליקטים רבים.

לדוגמה:

נוכל להכפיל את ה-fitness של כל לוח במספר המקסימאלי של התנגשויות מבין ההתנגשויות של המלכות.

* ניצור אוכלוסייה של לוחות, עליהם נפעיל את אלגוריתם minimal conflicts ובכל דור נבחר את מחצית הלוחות הטובים ביותר לדור הבא.
* נבצע את אלגוריתם Minimal conflicts ובכל פעם שניתקל במינימום לוקאלי נשתמש באחד מאופציות המוטציה שמומשו בסעיפים הקודמים.

1. **!!! יש לעשות מאפס !!!**