**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:

* בוצע על המחרוזת 'Hello World!'
* סך איטרציות מקסימאלי בכל טסט – 600.
* גדלי אוכלוסייה שנבדקו – 30, 100, 200, 500, 1000, 2000.
* סוגי היוריסטיקה – בול פגיעה, מרחק-אותיות.
* עבור כל קומבינציה (גודל אוכלוסייה, היוריסטיקה) ביצענו 100 טסטים.

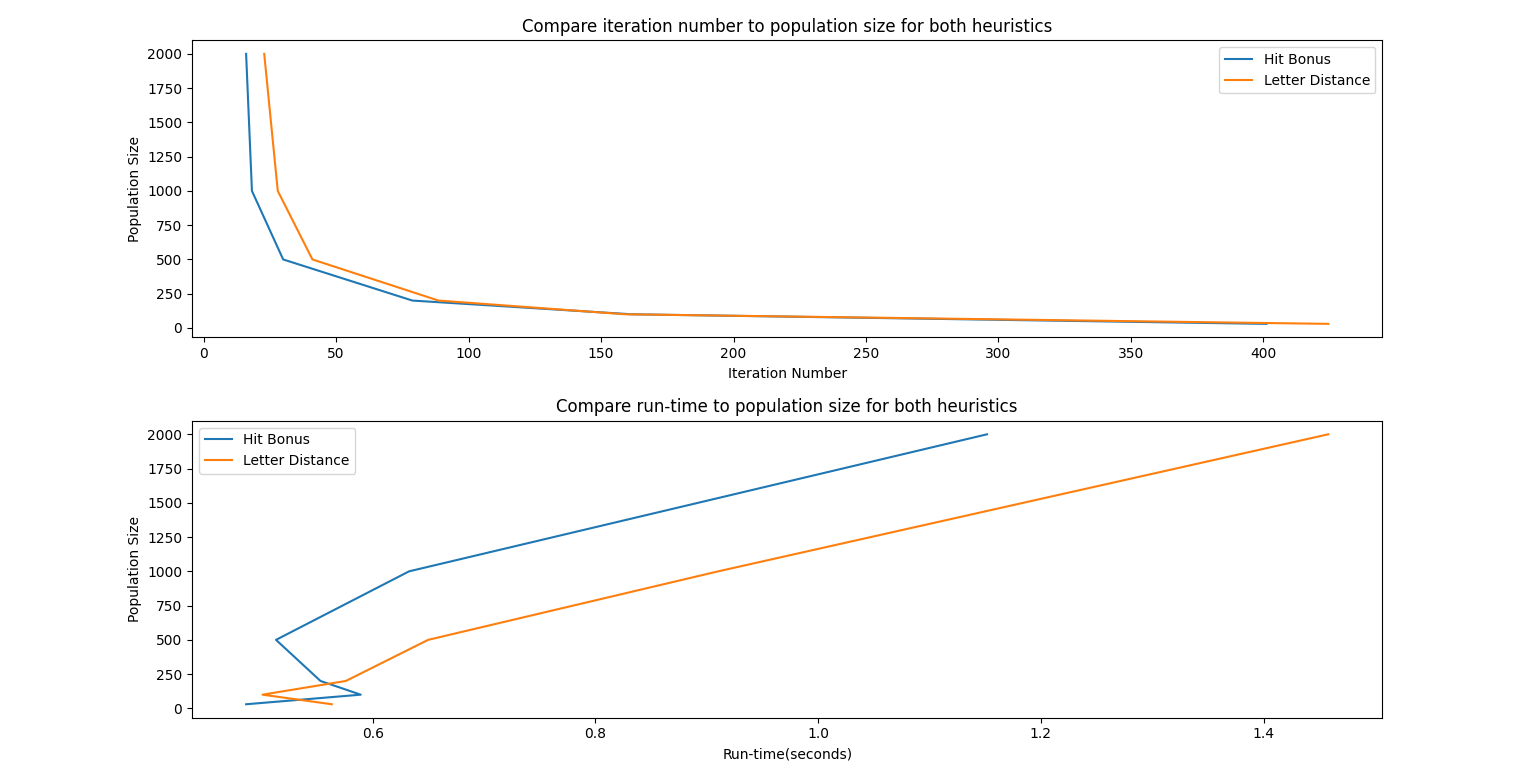
ניתן לראות מהגרפים המצורפים כי קיים trade-off ברור בין כמות איטרציות אשר קטן ככל שהאוכלוסייה גדלה לבין זמן הריצה.

הדבר נובע, מחישובים מרובים בכל איטרציה, לכן לקח יותר זמן לכל איטרציה, אך הגעה לפתרון בפחות איטרציות.

כאשר בדקנו על אוכלוסייה בגודל 30:

* עבור היוריסטיקת Hit Bonus הגענו לתוצאה רק ב-66% מההרצות.
* עבור היוריסטיקת מרחק-אותיות הגענו לתוצאה רק ב-61% מההרצות.

עבור אוכלוסיות גדולות יותר הגענו לתוצאות תמיד.



ניתן לראות שעבור אוכלוסיות גדולות יותר היוריסטיקת Hit Bonus אפקטיבית ומתכנסת מהר יותר מאשר היוריסטיקה מרחק-אותיות. ניתן לראות זאת, גם במס' האיטרציות הממוצע הלוקח לה להתכנס וגם בסך זמן הריצה. נציין כי באוכלוסיות קטנות יותר ההבדל קיים אך אינו משמעותי.

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.

* **אור-אל צריכה להוסיף משהו**

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.

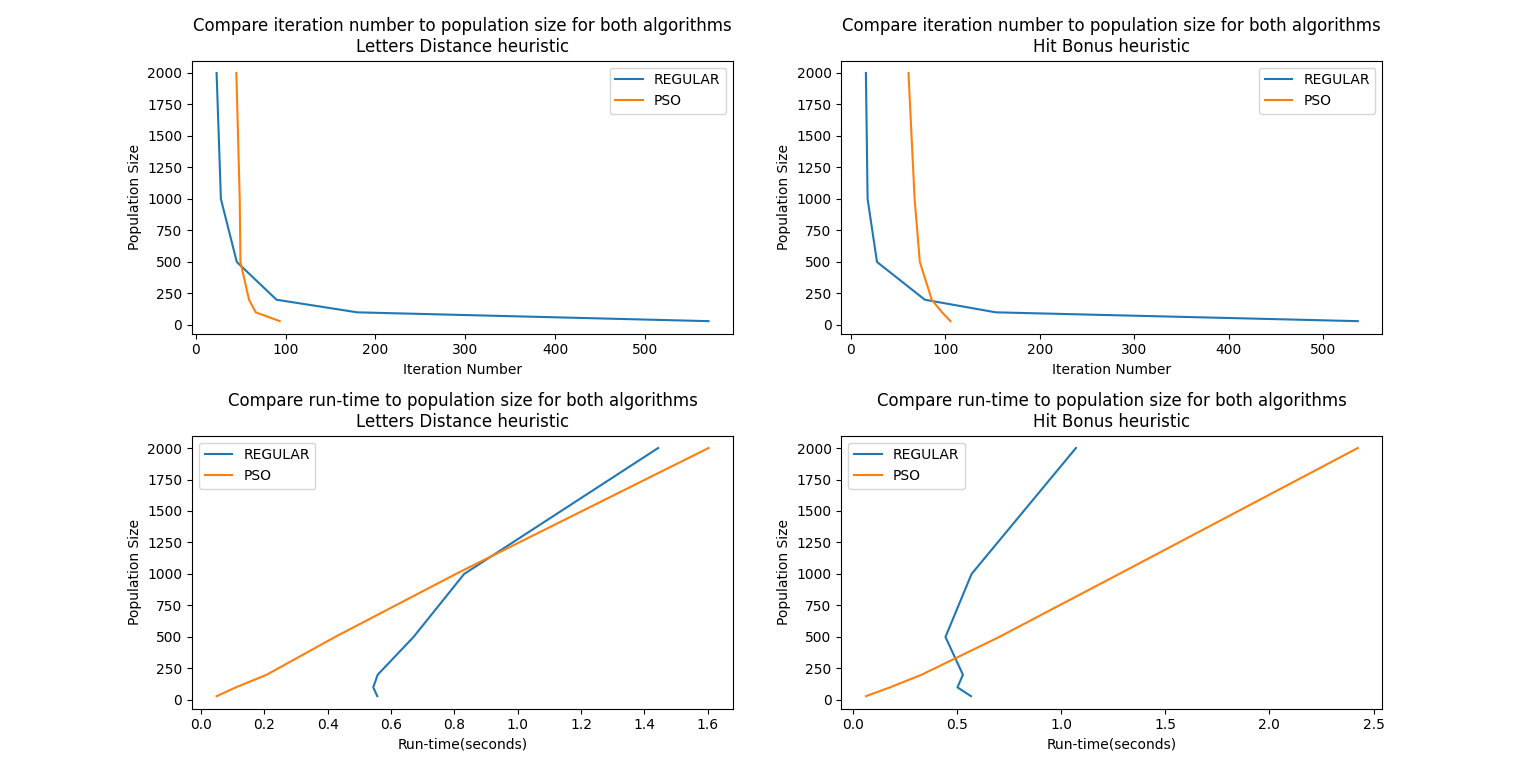
* בחירת הגנים הזכאים לריבוי נעשת (טרם הוספת אופציות זכאות נוספות) בצורה כזאת שנבחרים רק ממחצית הדור בעלי ה-fitness הגובה ביותר. סוג בחירה כזאת מגדיל את ה-EXPLOITION שכן מנצל את הגנים החזקים ביותר עד כה.

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. השוואת אלגוריתם PSO והאלגוריתם הגנטי ביחס להיוריסטיקת "בול פגיעה" והיוריסטיקת מרחק-אותיות:

* בוצע על המחרוזת 'Hello World!'
* סך איטרציות מקסימאלי בכל טסט – 2000.
* גדלי אוכלוסייה שנבדקו – 30, 100, 200, 500, 1000, 2000.
* סוגי היוריסטיקה – בול פגיעה, מרחק-אותיות.
* עבור כל קומבינציה (גודל אוכלוסייה, אלגוריתם, היוריסטיקה) ביצענו 100 טסטים.



ניתן לראות מבחינת זמני ריצה ביחס לסך האיטרציות כי היוריסטיקה מרחק-אותיות עבור שני האלגוריתמים עדיפה מאשר היוריסטיקת "בול פגיעה". נשים לב, עבור אוכלוסייה גדולה מאוד היוריסטיקת "בול פגיעה" עם אלגוריתם PSO רצה בממוצע לאיטרציה 2 שניות בעוד שהיוריסטיקת מרחק-אותיות רצה בפחות מ-2 שניות עבור שני האלגוריתמים.

בנוסף, ניתן לראות כי כמות האיטרציות להתכנסות באלגוריתם PSO נראה די קבוע גם כאשר גודל האוכלוסייה משתנה, עבור היוריסטיקת מרחק-אותיות סובב סביב ה-70 איטרציות ועבור היוריסטיקת "בול פגיעה" סובב סביב ה-100 איטרציות.

נציין כי עבור אלגוריתם PSO קיים חולשה למינימום לוקאלי ככל שהאוכלוסייה קטנה.

על כן, איכות הפתרון פחותה לאלגוריתם זה ביחס לאלגוריתם הגנטי.

נציג אחוזי הצלחה (מתוך 2000 איטרציות) להגעה לפתרון הרצוי ולא למינימום לוקאלי:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| גודל האוכלוסייה/  אלגוריתם-היוריסטיקה | PSO +  Letters Distance | PSO +  Hit Bonus | Genetic Algorithm +  Letters Distance | Genetic Algorithm +  Hit Bonus |
| 2000 | 100% | 88% | 100% | 100% |
| 1000 | 96% | 81% | 100% | 100% |
| 500 | 91% | 75% | 100% | 100% |
| 200 | 80% | 48% | 100% | 100% |
| 100 | 70% | 25% | 100% | 100% |
| 30 | 32% | 2% | 100% | 100% |

ניתן לראות כי האלגוריתם הגנטי הינו מבצע יותר EXPLORATION ובכך נמנע מהמינימום הלוקאלי, לעומת PSO אשר מגיע לפתרון בד"כ במספר קבוע של צעדים (כפי שניתן לראות בגרפים לעיל) אך עם סיכוי גבוה יותר שפתרון זה יהיה פחות איכותי, במיוחד אם גודל האוכלוסייה קטן, שכן כפי שראינו ה-EXPLORATION של PSO בא כתוצאה מאוכלוסייה גדולה.

**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 !!!**