מעבדה בבינה מלאכותית

דו"ח תרגיל בית 1 – Genetic Algorithms

<u>שמות</u>:

אסיל נחאס, 212245096

עוביידה חטיב, 201278066

סביבת ההרצה והכלים בהם נעשה שימוש

- המפענח שהשתמשנו בו: GNU bash 3.2.57
- גרסת הפייתון: 3.9.6
- חבילות לא סטנדטיות שהשתמשנו בהם: matplotlib 3.9.4

הרצת קובץ ה- EXE

עם תחילת ההרצה, המשתמש יתבקש לבחור את האופציה הרצויה עבור כל פרמטר מבין האפשרויות המוצגות עבורו, או ללחוץ Enter במקרה שהוא מעוניין להשתמש בערך ברירת המחדל שנקבע על ידינו. הפרמטרים שהשתמש יתבקש לבחור:

- סוג השיחלוף.
- שיטת חישוב הפיטניס.
- שיטת בחירת ההורים.
- זמן הריצה המקסימלי של האלגוריתם (בשניות).
- הזנת קלט לאלגוריתם: שלב המורכב משני שלבים:
  - אופן הזנת הקלט: לבחור אם הקלט יוזן ידנית או באמצעות קובץ.
  - הזנת הקלט: <u>במקרה שהקלט מוכנס ידנית</u>, המשתמש יתבקש להכניס את הגינום ההתחלתי במקרה שיש, ובמקרה שאין ללחוץ Enter. לאחר מכן, הוא יתבקש להכניס את גינום המטרה במקרה שיש. אם לא יוכנס ערך, גינום המטרה יהיה "Hello world!" שהוא ברירת המחדל.

    <u>במקרה שהקלט מוזן דרך קובץ</u>, המשתמש יתבקש להקליד את הנתיב לקובץ. הקובץ צריך להיות באחד הפורמטים: txt, json, jsonl. כמו כן, הוא צריך להיות מאותו מבנה של קבצי הבעיות בסעיפים 11 ו- 12.

אם המשתמש מזין גנום התחלתי, כל פרטי האוכלוסייה יאותחלו כנושאים אותו גנום, אחרת פרטי האוכלוסייה יאותחלו באופן רנדומלי.

הערה: במקרה בו הקובץ מכיל יותר מבעיה אחת, האלגוריתם יחלץ את הנתונים של הבעיה הראשונה מביניהם. מאחר וההנחיות של תרגיל הבית אינן מבהירות כיצד להתמודד עם הקבצים, הנחנו כי הרצה אחת של האלגוריתם אמורה לטפל ולספק תוצאות לבעיה אחת.

בכל שלב, אם הבחירה לא תואמת לאפשרויות שהוצגו, תתקבל הודעה בהתאם, והמשתמש יידרש להזין שוב. במקרה שהקובץ אינו בפורמט הצפוי התוכנית מסתיימת עם הודעת שגיאה.

האלגוריתם מזהה את סוג הבעיה (Target String, Matrix Transform או Bin Packing) באופן אוטומטי, לפי תוכן הקלט – ואין צורך שהמשתמש יציין אותה בעצמו (למשל, קובץ TXT יצביע על בעיה

להלן דוגמה הרצה עם תיאור הבחירות שנעשו.

```
Select crossover type:
  1. SINGLE
  2. TWO
  3. UNIFORM
Choose one of the options or Enter for default: 3
Selected: UNIFORM

Select fitness mode:
  1. DISTANCE
  2. LCS
  3. BINS_DIFF
Choose one of the options or Enter for default:
Using default: LCS

Select parent selection method:
  1. TOP_HALF_UNIFORM
  2. RWS
  3. SUS
  4. TOURNAMENT_DET
  5. TOURNAMENT_STOCH
Choose one of the options or Enter for default: 4
Selected: TOURNAMENT_DET

Enter the maximum time for the algorithm to run (in secs):
Max time: 60

Would you like to enter the initial genome/matrix or read it from a file?
  1. Enter manually
  2. Read from a file
Choose 1/2 or Enter for default: 1

Enter initial genome/matrix (or Enter if there's none):

Enter target genome/matrix (or Enter for default): Hello and Welcome!
```

המשתמש בחר בשיטת השיחלוף UNIFORM, לא בחר בשיטת חישוב פיטניס ועל כן נבחר עבורו השיטה LCS שהגדרנו כברירת המחדל, בחר בטורניר דטרמיניסטי כשיטת בחירת הורים, בחר שהזמן המקסימלי לריצת האלגוריתם יהיה 60 שניות, בחר להזין את הקלט באופן ידני כאשר לא הזין גינום התחלתי ולכן האוכלוסייה ההתחלתי תבחר עבורו באופן אקראי, ולבסוף בחר במחרוזת "Hello and Welcome!" כמחרוזת היעד.

הרצת קובץ הפייתון

הקוד ניתן להרצה דרך שורת הפקודות תוך קבלת ארגומנט של מגבלת זמן הריצה בשניות.
לדוגמה:

python lab1.py 60

ניתן להריץ את הקוד במספר צורות בהתאם לאם הגנומים ההתחלתי הסופי ניתנים, ולאופן בו הם ניתנים:

1. **בלי לקבל את גינומים התחלתי יעד**: במקרה זה האלגוריתם יאתחל את האוכלוסייה בגנומים רנדומליים, וגנום היעד יהיה מחרוזת ברירת המחדל " Hello world!". במקרה הזה ההרצה תהיה באמצעות הפקודה:

```
python script.py <max_time>
```

2. **עם גינום היעד בלבד**: האלגוריתם יאתחל את האוכלוסייה בגנומים רנדומליים. פרט היעד יהיה הניתן כארגומנט. במקרה הזה ההרצה תהיה באמצעות הפקודה:

```
python script.py <max_time> <target_genome>
```

3. **עם גנומים התחלתי ויעד**: האוכלוסייה תאותחל בפרטים בעלי הגנום ההתחלתי הניתן כארגומנט. פרט היעד יהיה הניתן כארגומנט. הגינומים (ההתחלתי והיעד) יכולים להינתן כארגומנטים במפורש, באמצעות קובץ TXT בפורמט התואם לקבצי הבעיה בסעיף 11, או באמצעות קובץ JSON בפורמט התואם לקבצי הבעיה בסעיף 12, ואז האלגוריתם יחלץ אותם משם. הפורמט של פקודות שני המקרים הם:

```
python script.py <max_time> <initial_genome> <target_genome>

python script.py <max_time> <file_path>
```

הערה: במקרה בו הקובץ מכיל יותר מבעיה אחת, האלגוריתם יחלץ את הנתונים של הבעיה הראשונה מביניהם. מאחר וההנחיות של תרגיל הבית אינן מבהירות כיצד להתמודד עם הקבצים, הנחנו כי הרצה אחת של האלגוריתם אמורה לטפל ולספק תוצאות לבעיה אחת.

ייצוג הפרטים והאוכלוסייה

בקוד שלנו, הפרטים מיוצגים באמצעות מופעים של המחלקה Individual, השומרת עבור כל פרט את הגינום ואת הפיטניס שלו. כמו כן, המחלקה מכילה מתודה המחשבת ומעדכנת את משתנה הפיטניס של הפרט.

האוכלוסייה מיוצגת ע"י מופע של המחלקה Population, המכילה משתנים עבור גודל האוכלוסייה, מחרוזת המטרה, ורשימת הפרטים של האוכלוסייה. בנוסף, המחלקה מכילה את המתודות הבאות: מתודה המאתחלת אוכלוסייה כקבוצה של פרטים עם גנומים אקראיים באורך מחרוזת המטרה, מתודה המחשבת ומעדכנת את הפיטניס של כל אחד מפרטי

האוכלוסייה, המתודה הממיינת את פרטי האוכלוסייה לפי ערכי הפיטניס שלהם, ומתודה המעבירה את הפרטים עם הפיטניס הכי טוב לדור הבא (Elitism).

להלן המבנים ההתחלתיים של שתי המחלקות, אליהם יתווספו עוד משתנים ומתודות לפי הנדרש במעבדה.

```
class Individual:
    def __init__(self, genome):
        self.genome = genome
        self.fitness = None
    def calculate_fitness(self, target)

class Population:
    def __init__(self, size, target):
        self.size = size
        self.target = target
        self.individuals = self.init_population()
    def init_population(self)
    def update_fitness(self)
    def sort_by_fitness(self)
    def elitism(self, buffer, esize)
```

בחירת הפרמטרים

תנאי העצירה בשל התכנסות מקומית נבחר להיות כאשר הפיטניס הטוב ביותר לא משתפר במשך 50 איטרציות. ההחלטה מתבססת על תצפיות שלנו שהראו כי אם לא מתרחש שיפור בתוך 50 דורות, אז לא יתרחש גם לאחר 100,200 ואפילו 400 איטרציות נוספות. ההחלטה אודות תנאי העצירה התקבלה עוד לפני בחירת מספר הפרטים באוכלוסייה, ולכן נלקחו בחשבון כל גדלי האוכלוסייה הפוטנציאלים, תוך ביצוע 100 הרצאות עבור כל אחד מהגדלים.

תנאי עצירה נוספים הם התכנסות גלובלית והגעה לסף הזמן הנתון כקלט. התכנסות גלובלית מתרחשת כאשר ערך הפיטניס של הפתרון הטוב ביותר בדור כלשהו שווה ל- 0. הגעה לסף הזמן נבדקת בסיום כל דור, ואם הזמן שחלף מאז תחילת הריצה הוא יותר מזה שניתן כארגומנט האלגוריתם נעצר.

```
Population size: 1024
  Success rate: 0.02
  No convergence after 50 iterations: 98
  No convergence after 100 iterations: 98
  No convergence after 200 iterations: 98
  No convergence after 400 iterations: 98
Population size: 2048
  Success rate: 0.25
  No convergence after 50 iterations: 75
  No convergence after 100 iterations: 75
  No convergence after 200 iterations: 75
  No convergence after 400 iterations: 75
Population size: 4096
  Success rate: 0.81
  No convergence after 50 iterations: 19
  No convergence after 100 iterations: 19
  No convergence after 200 iterations: 19
  No convergence after 400 iterations: 19
Population size: 8192
  Success rate: 0.97
  No convergence after 50 iterations: 3
  No convergence after 100 iterations: 3
  No convergence after 200 iterations: 3
  No convergence after 400 iterations: 3
```

המספר המקסימלי של דורות הוחלט להיות 120. ערך זה גדול מכל מספר דורות מקסימלי שהתקבל ב- 400 הרצות שהתבצעו כמתואר מקודם, ולכן הוא מבטיח באופן כמעט וודאי שהאלגוריתם לא ייעצר בשל מספר לא מספיק של דורות.

```
Population size: 1024
  Success rate: 0.02
  Max generations: 102
Population size: 2048
  Success rate: 0.23
  Max generations: 89
Population size: 4096
  Success rate: 0.77
  Max generations: 84
Population size: 8192
  Success rate: 1.00
  Max generations: 33
```

בחירת **גודל האוכלוסייה** נעשתה על סמך ניתוח סטטיסטי שנעשה על 100 הרצות לכל גודל אוכלוסייה פוטנציאלי. המטרה הייתה לבחור את האוכלוסייה הקטנה ביותר המתכנסת גלובלית בלפחות 95% מהמקרים. הגודל הראשון שענה על הקריטריון הוא 2048 ועל כן נבחר.

סעיף 1

הוספנו למחלקה Population מתודה אשר מחשבת ומדפיסה את הפרט עם ה- Fitness הגבוה ביותר, הפרט עם ה- Fitness הנמוך ביותר, ממוצע, סטיית תקן, וטווח ערכי הפיטניס של האוכלוסייה.

```
def generation_stats_update(self, generation):
    best_fit  = self.individuals[0].fitness
    worst_fit = self.individuals[-1].fitness
    fitness_range = worst_fit - best_fit
    sum_fit   = sum(ind.fitness for ind in self.individuals)
    avg_fit   = sum_fit / self.size
    variance  = sum((ind.fitness - avg_fit)**2 for ind in self.individuals)
                    / self.size
    std_dev   = math.sqrt(variance) if variance>0 else 0

    print(f"Gen{generation}."
            f" Best: {self.individuals[0].genome} ({best_fit})",
            f" Worst: {self.individuals[-1].genome} ({worst_fit}) ",
            f" Fitness Range: {fitness_range} ",
            f" Avg: {avg_fit:.2f} ",
            f" Std: {std_dev:.2f} ")
```

מצורפות תוצאות של 3 הרצות שונות שנעשו:

```
Gen0. Best: b]`\k4xxsfS6 (141)  Worst: )H'M%v"*?4/o (712)   Fitness Range: 571   Avg: 421.53   Std: 80.65
Gen1. Best: A[an[5noepg) (108)  Worst: gGQ;@[/0Y3Bn (571)   Fitness Range: 463   Avg: 354.21   Std: 62.65
Gen2. Best: A[an[5noepg) (108)  Worst: stEI=U$Ml['y (524)   Fitness Range: 416   Avg: 304.48   Std: 55.84
Gen3. Best: rchgr joooh. (92)  Worst: y*vA8c9NMC^% (466)   Fitness Range: 374   Avg: 261.66   Std: 51.03
Gen4. Best: Secpq#tqvbh" (53)  Worst: ^^+k%(pU\)mk (382)   Fitness Range: 329   Avg: 223.53   Std: 45.57
Gen5. Best: Secpq#tqvbh" (53)  Worst: VN$xH,'@QqG$ (369)   Fitness Range: 316   Avg: 191.01   Std: 42.40
Gen6. Best: Icdpg yprj`& (37)  Worst: DgZNC&pbD*3< (312)   Fitness Range: 275   Avg: 161.97   Std: 37.99
Gen7. Best: Hklmo#oqomf! (26)  Worst: (OosiRsVnDhw (283)   Fitness Range: 257   Avg: 137.22   Std: 33.75
Gen8. Best: Hklmo#oqomf! (26)  Worst: 1rrr^.yYgb%} (279)   Fitness Range: 253   Avg: 116.76   Std: 30.29
Gen9. Best: Hklmo#oqomf! (26)  Worst: hvtlqzoq_uY4 (217)   Fitness Range: 191   Avg: 99.99   Std: 27.63
Gen10. Best: Hgjoo yprjf! (14)  Worst: =Zej\rxyZVa' (198)   Fitness Range: 184   Avg: 86.16   Std: 26.13
Gen11. Best: Hgjoo yprjf! (14)  Worst: HeOmp2u\xcnp (174)   Fitness Range: 160   Avg: 73.81   Std: 24.45
Gen12. Best: Hgjoo ypslf! (13)  Worst: M\hmo|oouuG! (160)   Fitness Range: 147   Avg: 63.80   Std: 23.53
Gen13. Best: Eelmm wmrkb! (11)  Worst: Q`dpgzsnglT! (156)   Fitness Range: 145   Avg: 55.38   Std: 23.05
Gen14. Best: Helmo tnsmf! (9)  Worst: =knup*yosfXp (139)   Fitness Range: 130   Avg: 48.02   Std: 22.49
Gen15. Best: Helmo tnsmf! (9)  Worst: Ftlmj|oooub! (137)   Fitness Range: 128   Avg: 41.30   Std: 21.70
Gen16. Best: Hdlmo wornf! (6)  Worst: @enup|ypgla" (130)   Fitness Range: 124   Avg: 36.22   Std: 21.95
Gen17. Best: Helmo yorkc! (5)  Worst: Hdcgjztospa" (122)   Fitness Range: 117   Avg: 31.41   Std: 21.48
Gen18. Best: Helmo yorkc! (5)  Worst: Helmo|oqrj[! (114)   Fitness Range: 109   Avg: 27.41   Std: 21.16
Gen19. Best: Helmo yorkc! (5)  Worst: Iejgj#vpqlfy (109)   Fitness Range: 104   Avg: 24.36   Std: 21.24
Gen20. Best: Helmo yorld! (3)  Worst: Haomo}wponc! (108)   Fitness Range: 105   Avg: 21.52   Std: 20.62
Gen21. Best: Helmo yorld! (3)  Worst: Gclop ypqm_{ (107)   Fitness Range: 104   Avg: 19.56   Std: 20.95
Gen22. Best: Helmo world" (2)  Worst: Hgkkm|vpqic! (105)   Fitness Range: 103   Avg: 18.09   Std: 20.53
Gen23. Best: Helmo world! (1)  Worst: Hemmo|yprjf! (101)   Fitness Range: 100   Avg: 17.18   Std: 21.15
Gen24. Best: Helmo world! (1)  Worst: Ielmo}wornf! (99)   Fitness Range: 98   Avg: 15.90   Std: 20.71
Gen25. Best: Helmo world! (1)  Worst: Helmp|ypqlc" (100)   Fitness Range: 99   Avg: 14.98   Std: 20.75
Gen26. Best: Helmo world! (1)  Worst: Helmo}vpqlf! (99)   Fitness Range: 98   Avg: 14.21   Std: 20.93
Gen27. Best: Hello world! (0)  Worst: Helmo}vprlf! (98)   Fitness Range: 98   Avg: 13.94   Std: 21.29
Global optimum found!
Finished after 28 generations.
```

```
Gen0. Best: LOqnP _nrOt# (136)  Worst: "1+"lF**!,#p (705)   Fitness Range: 569   Avg: 419.53   Std: 80.58
Gen1. Best: bkuho!dotye7 (103)  Worst: %P1 _PZ8+7.[ (575)   Fitness Range: 472   Avg: 353.14   Std: 62.62
Gen2. Best: bkuho!dotye7 (103)  Worst: DTd&abG/+f(i (500)   Fitness Range: 397   Avg: 303.44   Std: 56.03
Gen3. Best: Qkuho!dotye7 (86)  Worst: 'u+F&Au 9Xi3 (439)   Fitness Range: 353   Avg: 259.95   Std: 50.24
Gen4. Best: Eapci0oimve# (74)  Worst: FnwGuN5k#)as (412)   Fitness Range: 338   Avg: 222.80   Std: 45.64
Gen5. Best: JXbko#xotye0 (61)  Worst: )WI\oYX v-3$ (382)   Fitness Range: 321   Avg: 190.44   Std: 41.17
Gen6. Best: FZomr*wtqjU  (54)  Worst: 3gaS0}zmx<@. (323)   Fitness Range: 269   Avg: 162.13   Std: 37.17
Gen7. Best: J[l]o!ypmcg! (48)  Worst: Uyl>TzXyvfj_ (315)   Fitness Range: 267   Avg: 138.04   Std: 33.73
Gen8. Best: D`loq"tovqi) (41)  Worst: :Hj^f0uMJ"b3 (254)   Fitness Range: 213   Avg: 117.65   Std: 29.83
Gen9. Best: Jgrkp!xsrli" (24)  Worst: H]unPsttbmUS (223)   Fitness Range: 199   Avg: 101.43   Std: 28.33
Gen10. Best: Jgrkp!xsrli" (24)  Worst: Pwq<[$bwNnM% (197)   Fitness Range: 173   Avg: 86.86   Std: 25.35
Gen11. Best: Gdqho!tlpke" (23)  Worst: .dqnozdoylv6 (189)   Fitness Range: 166   Avg: 75.80   Std: 24.66
Gen12. Best: Heolo#xowlf% (18)  Worst: D`lViqqtqJe! (165)   Fitness Range: 147   Avg: 65.53   Std: 22.97
Gen13. Best: Iemmo!tlpke" (15)  Worst: Cawwf{yhdkg! (158)   Fitness Range: 143   Avg: 57.32   Std: 22.01
Gen14. Best: Ieomo yoqld$ (11)  Worst: Hnwmr%rovqW| (147)   Fitness Range: 136   Avg: 50.06   Std: 21.41
Gen15. Best: Ieomo!yorkd  (10)  Worst: Fgdho|tmund3 (136)   Fitness Range: 126   Avg: 44.75   Std: 21.65
Gen16. Best: Jeono wormd" (9)  Worst: Beodo|korhi# (132)   Fitness Range: 123   Avg: 39.76   Std: 21.50
Gen17. Best: Jeono wormd" (9)  Worst: Caqho{yhql^! (125)   Fitness Range: 116   Avg: 35.18   Std: 21.10
Gen18. Best: Hgnlo xorld" (6)  Worst: Jgrkp(ulppd} (123)   Fitness Range: 117   Avg: 31.86   Std: 21.10
Gen19. Best: Iello workd  (3)  Worst: Egkmn}yuqha  (118)   Fitness Range: 115   Avg: 28.15   Std: 20.85
Gen20. Best: Iello workd  (3)  Worst: Hlloo|wovli$ (114)   Fitness Range: 111   Avg: 25.90   Std: 21.26
Gen21. Best: Iello workd  (3)  Worst: Geoko|xlspe" (108)   Fitness Range: 105   Avg: 22.72   Std: 20.37
Gen22. Best: Hello workd  (2)  Worst: Ggnlo}yotli! (107)   Fitness Range: 105   Avg: 20.50   Std: 20.73
Gen23. Best: Hello workd  (2)  Worst: Idloo}xotmd  (103)   Fitness Range: 101   Avg: 18.81   Std: 20.77
Gen24. Best: Hello worle! (1)  Worst: Hdloo}wovld" (102)   Fitness Range: 101   Avg: 17.76   Std: 21.01
Gen25. Best: Hello world! (0)  Worst: Iemmn|xormc  (100)   Fitness Range: 100   Avg: 15.95   Std: 20.77
Global optimum found!
```

```
Gen0. Best: b]`\k4xxsfS6 (141)  Worst: )H'M%v"*?4/o (712)   Fitness Range: 571   Avg: 421.53   Std: 80.65
Gen1. Best: A[an[5noepg) (108)  Worst: gGQ;@[/0Y3Bn (571)   Fitness Range: 463   Avg: 354.21   Std: 62.65
Gen2. Best: A[an[5noepg) (108)  Worst: stEI=U$Ml['y (524)   Fitness Range: 416   Avg: 304.48   Std: 55.84
Gen3. Best: rchgr joooh. (92)  Worst: y*vA8c9NMC^% (466)   Fitness Range: 374   Avg: 261.66   Std: 51.03
Gen4. Best: Secpq#tqvbh" (53)  Worst: ^^+k%(pU\)mk (382)   Fitness Range: 329   Avg: 223.53   Std: 45.57
Gen5. Best: Secpq#tqvbh" (53)  Worst: VN$xH,'@QqG$ (369)   Fitness Range: 316   Avg: 191.01   Std: 42.40
Gen6. Best: Icdpg yprj`& (37)  Worst: DgZNC&pbD*3< (312)   Fitness Range: 275   Avg: 161.97   Std: 37.99
Gen7. Best: Hklmo#oqomf! (26)  Worst: (OosiRsVnDhw (283)   Fitness Range: 257   Avg: 137.22   Std: 33.75
Gen8. Best: Hklmo#oqomf! (26)  Worst: 1rrr^.yYgb%} (279)   Fitness Range: 253   Avg: 116.76   Std: 30.29
Gen9. Best: Hklmo#oqomf! (26)  Worst: hvtlqzoq_uY4 (217)   Fitness Range: 191   Avg: 99.99   Std: 27.63
Gen10. Best: Hgjoo yprjf! (14)  Worst: =Zej\rxyZVa' (198)   Fitness Range: 184   Avg: 86.16   Std: 26.13
Gen11. Best: Hgjoo yprjf! (14)  Worst: HeOmp2u\xcnp (174)   Fitness Range: 160   Avg: 73.81   Std: 24.45
Gen12. Best: Hgjoo ypslf! (13)  Worst: M\hmo|oouuG! (160)   Fitness Range: 147   Avg: 63.80   Std: 23.53
Gen13. Best: Eelmm wmrkb! (11)  Worst: Q`dpgzsnglT! (156)   Fitness Range: 145   Avg: 55.38   Std: 23.05
Gen14. Best: Helmo tnsmf! (9)  Worst: =knup*yosfXp (139)   Fitness Range: 130   Avg: 48.02   Std: 22.49
Gen15. Best: Helmo tnsmf! (9)  Worst: Ftlmj|oooub! (137)   Fitness Range: 128   Avg: 41.30   Std: 21.70
Gen16. Best: Hdlmo wornf! (6)  Worst: @enup|ypgla" (130)   Fitness Range: 124   Avg: 36.22   Std: 21.95
Gen17. Best: Helmo yorkc! (5)  Worst: Hdcgjztospa" (122)   Fitness Range: 117   Avg: 31.41   Std: 21.48
Gen18. Best: Helmo yorkc! (5)  Worst: Helmo|oqrj[! (114)   Fitness Range: 109   Avg: 27.41   Std: 21.16
Gen19. Best: Helmo yorkc! (5)  Worst: Iejgj#vpqlfy (109)   Fitness Range: 104   Avg: 24.36   Std: 21.24
Gen20. Best: Helmo yorld! (3)  Worst: Haomo}wponc! (108)   Fitness Range: 105   Avg: 21.52   Std: 20.62
Gen21. Best: Helmo yorld! (3)  Worst: Gclop ypqm_{ (107)   Fitness Range: 104   Avg: 19.56   Std: 20.95
Gen22. Best: Helmo world" (2)  Worst: Hgkkm|vpqic! (105)   Fitness Range: 103   Avg: 18.09   Std: 20.53
Gen23. Best: Helmo world! (1)  Worst: Hemmo|yprjf! (101)   Fitness Range: 100   Avg: 17.18   Std: 21.15
Gen24. Best: Helmo world! (1)  Worst: Ielmo}wornf! (99)   Fitness Range: 98   Avg: 15.90   Std: 20.71
Gen25. Best: Helmo world! (1)  Worst: Helmp|ypqlc" (100)   Fitness Range: 99   Avg: 14.98   Std: 20.75
Gen26. Best: Helmo world! (1)  Worst: Helmo}vpqlf! (99)   Fitness Range: 98   Avg: 14.21   Std: 20.93
Gen27. Best: Hello world! (0)  Worst: Helmo}vprlf! (98)   Fitness Range: 98   Avg: 13.94   Std: 21.29
Global optimum found!
Finished after 28 generations.
```

סעיף 2

חישוב והדפסת שני זמני הריצה, ה- Clock tick והאבסולוטי, התבצעו ע"י אתחול שני משתנים המחזיקים את הזמן של תחילת הביצוע :

```python
start_wall_time = time.time()
start_cpu_time  = time.process_time()
```

בנוסף לפונקציה הבאה המחשבת ומדפיסה את הזמן שעבר מאז תחילת ההרצה:

```python
def time_compute(start_cpu_time, start_wall_time):
    ticks_cpu = time.process_time() - start_cpu_time
    elapsed   = time.time() - start_wall_time
    print(f"    Ticks CPU: {ticks_cpu:.4f}, Elapsed:
          {elapsed:.2f}s")
```

להלן המספרים שהתקבלו מאחת הריצות:

```
Gen0. Best: KZbsQ$qusp.! (136)  Worst: ##'!*sH;\($R (701)   Fitness Range: 565   Avg: 421.19   Std: 81.02
    Ticks CPU: 0.0419, Elapsed: 0.0421s
Gen1. Best: @^pVK:woykh$ (118)  Worst: .s70%TF &T8H (590)   Fitness Range: 472   Avg: 355.39   Std: 63.13
    Ticks CPU: 0.1760, Elapsed: 0.1774s
Gen2. Best: EaqVq:woyqK) (107)  Worst: tp CHfO<C/hV (537)   Fitness Range: 430   Avg: 304.44   Std: 56.10
    Ticks CPU: 0.3111, Elapsed: 0.3136s
Gen3. Best: D^ykU;woykh$ (93)  Worst: DLyw8G:1;r5} (470)   Fitness Range: 377   Avg: 261.95   Std: 50.25
    Ticks CPU: 0.4551, Elapsed: 0.4608s
Gen4. Best: ?qqZp nrxea' (79)  Worst: G:dhd`T5 Ubg (401)   Fitness Range: 322   Avg: 224.93   Std: 45.07
    Ticks CPU: 0.6023, Elapsed: 0.6116s
Gen5. Best: Lbael!woYmc" (57)  Worst: 4q)FK03r\om} (386)   Fitness Range: 329   Avg: 193.42   Std: 40.56
    Ticks CPU: 0.7361, Elapsed: 0.7464s
Gen6. Best: Ednpf tbyog% (52)  Worst: *(\Xd.JibSLy (356)   Fitness Range: 304   Avg: 166.14   Std: 36.91
    Ticks CPU: 0.8675, Elapsed: 0.8786s
Gen7. Best: 5doho&xrnec  (50)  Worst: IZ$v7,pc}:A! (277)   Fitness Range: 227   Avg: 142.76   Std: 33.43
    Ticks CPU: 1.0012, Elapsed: 1.0135s
Gen8. Best: Ednpf votmc" (25)  Worst: *\w'd$oikgov (256)   Fitness Range: 231   Avg: 122.70   Std: 31.12
    Ticks CPU: 1.1344, Elapsed: 1.1479s
Gen9. Best: Ednpf votmc" (25)  Worst: JDZsq:"oya\9 (223)   Fitness Range: 198   Avg: 104.98   Std: 28.43
    Ticks CPU: 1.2657, Elapsed: 1.2800s
Gen10. Best: Ednpf votmc" (25)  Worst: ATQ\j([gyiov (222)   Fitness Range: 197   Avg: 89.95   Std: 25.91
    Ticks CPU: 1.3984, Elapsed: 1.4140s
Gen11. Best: Ednpf votmc" (25)  Worst: G^Svg)Ws'km% (185)   Fitness Range: 160   Avg: 77.68   Std: 24.31
    Ticks CPU: 1.5310, Elapsed: 1.5476s
Gen12. Best: Edljn!ypqkc" (15)  Worst: Elxve}uurkj6 (171)   Fitness Range: 156   Avg: 67.73   Std: 23.41
    Ticks CPU: 1.6614, Elapsed: 1.6788s
Gen13. Best: Edljn!ypqkc" (15)  Worst: Ah}og tkyqrv (156)   Fitness Range: 141   Avg: 59.18   Std: 22.46
    Ticks CPU: 1.7932, Elapsed: 1.8119s
Gen14. Best: Edljn!ypqkc" (15)  Worst: CXegg' kqqn! (152)   Fitness Range: 137   Avg: 52.23   Std: 21.84
    Ticks CPU: 1.9245, Elapsed: 1.9441s
Gen15. Best: Hdllo!xookh  (12)  Worst: Sbael!woyoa| (140)   Fitness Range: 128   Avg: 46.45   Std: 21.21
    Ticks CPU: 2.0583, Elapsed: 2.0817s
Gen16. Best: Jfllo votmc" (9)  Worst: Edjix}qqurj! (134)   Fitness Range: 125   Avg: 41.91   Std: 21.52
    Ticks CPU: 2.1909, Elapsed: 2.2157s
Gen17. Best: Jfllo votmc" (9)  Worst: H[jrpywqtde$ (124)   Fitness Range: 115   Avg: 37.20   Std: 21.16
    Ticks CPU: 2.3277, Elapsed: 2.3565s
```

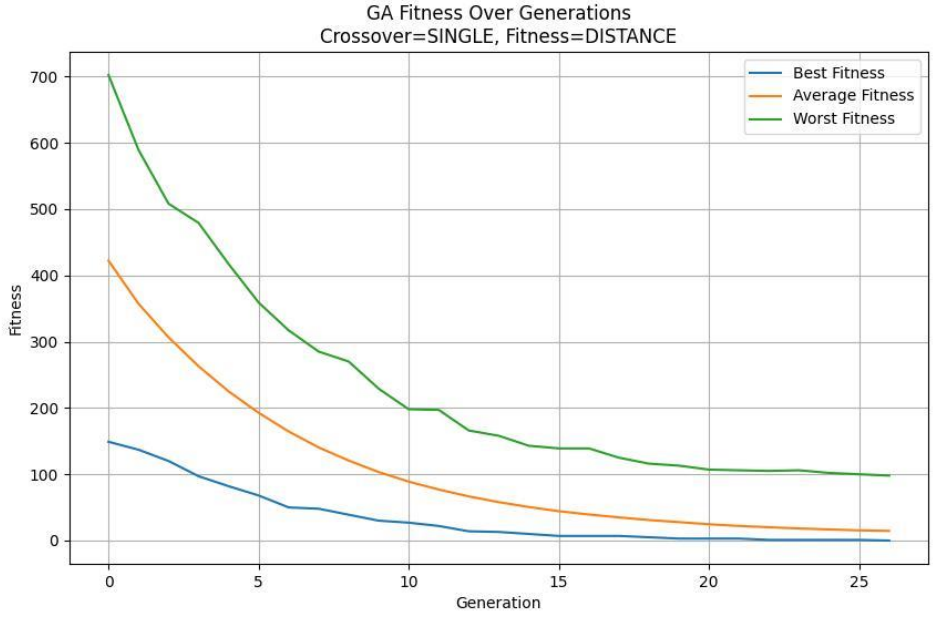
```
Gen18. Best: Hello votmc" (6)  Worst: Hhtio}vuspc  (121)   Fitness Range: 115   Avg: 33.12   Std: 20.86
    Ticks CPU: 2.4612, Elapsed: 2.4917s
Gen19. Best: Hello votmc" (6)  Worst: Ednlg wrrfc} (116)   Fitness Range: 110   Avg: 29.78   Std: 20.67
    Ticks CPU: 2.5927, Elapsed: 2.6240s
Gen20. Best: Hello wormc  (3)  Worst: Jgjjo|rqukc" (113)   Fitness Range: 110   Avg: 27.32   Std: 21.14
    Ticks CPU: 2.7254, Elapsed: 2.7596s
Gen21. Best: Hello wormc  (3)  Worst: Jfojo}wophh  (112)   Fitness Range: 109   Avg: 24.52   Std: 20.80
    Ticks CPU: 2.8582, Elapsed: 2.8939s
Gen22. Best: Hello wormc  (3)  Worst: Jelfozxurkc" (108)   Fitness Range: 105   Avg: 22.57   Std: 21.07
    Ticks CPU: 2.9893, Elapsed: 3.0259s
Gen23. Best: Hello wormc  (3)  Worst: Hbmjn}xoukc" (107)   Fitness Range: 104   Avg: 20.84   Std: 20.99
    Ticks CPU: 3.1216, Elapsed: 3.1597s
Gen24. Best: Hello wormd! (1)  Worst: Jfllo{voqk`! (101)   Fitness Range: 100   Avg: 19.17   Std: 21.06
    Ticks CPU: 3.2528, Elapsed: 3.2916s
Gen25. Best: Hello wormd! (1)  Worst: Fdlln}voqkc  (102)   Fitness Range: 101   Avg: 17.64   Std: 20.86
    Ticks CPU: 3.3842, Elapsed: 3.4238s
Gen26. Best: Hello world! (0)  Worst: Jfllo!wotkc} (100)   Fitness Range: 100   Avg: 16.39   Std: 20.87
    Ticks CPU: 3.5153, Elapsed: 3.5558s
Global optimum found!
Finished after 27 generations.
Total wall-clock time: 3.56s, Total CPU time: 3.52s
```

הדפסת שני הגרפים מומשה באמצעות שתי הפונקציות הבאות שהתווספו למחלקת Population, ואשר נקראות בתום ריצת האלגוריתם:

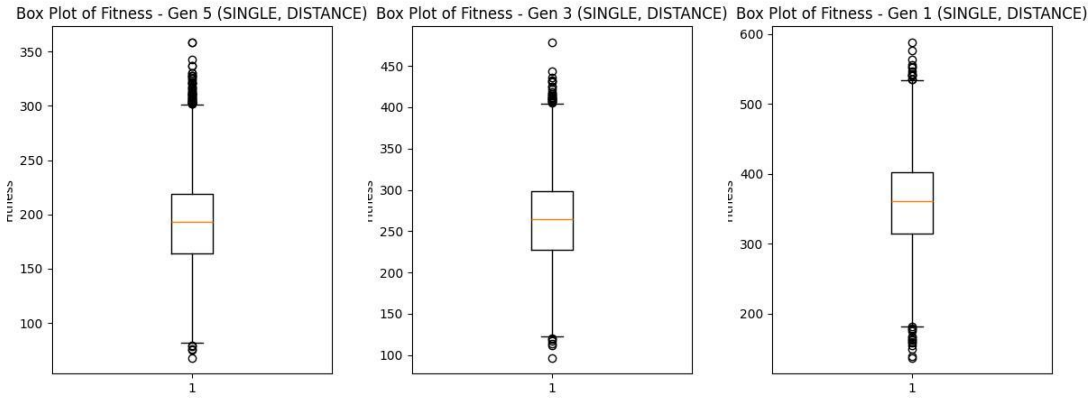
```
def fitness_plot(self):
    plt.figure(figsize=(10, 6))
    plt.plot(self.best_fitness_list,  label="Best Fitness")
    plt.plot(self.avg_fitness_list,   label="Average Fitness")
    plt.plot(self.worst_fitness_list, label="Worst Fitness")
    plt.title(f"GA Fitness Over Generations\nCrossover=
              {CROSSOVER_TYPE}, Fitness={FITNESS_MODE}")
    plt.xlabel("Generation")
    plt.ylabel("Fitness")
    plt.legend()
    plt.grid(True)
    plt.show()
```

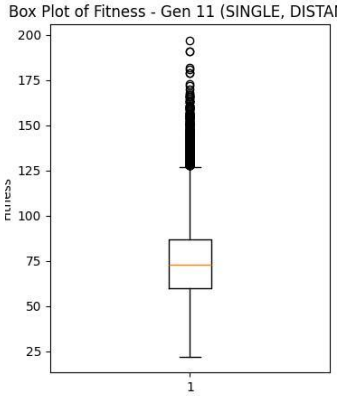
```
def fitness_boxplot(self):
    for g,data in enumerate(self.fitness_history):
        plt.figure(figsize=(4,5))
        plt.boxplot(data, showfliers=True)
        plt.title(f"Box Plot of Fitness - Gen {g}
            ({CROSSOVER_TYPE}, {FITNESS_MODE})")
        plt.ylabel("Fitness")
        plt.show()
```

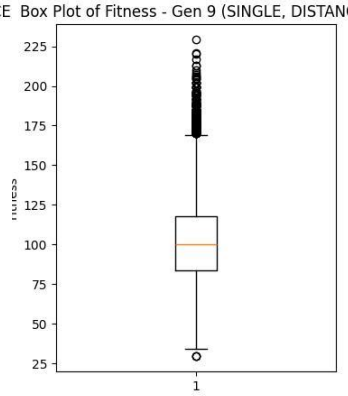
מצורף דוגמה לפלט שהתקבל בעקבות אחת ההרצות. ה- Boxplots, בשל היותם 27, צורפו חלקית, לסירוגין. בנוסף, לכל סוג של גרפים מצורף הסבר אודות מה הוא מבטא:

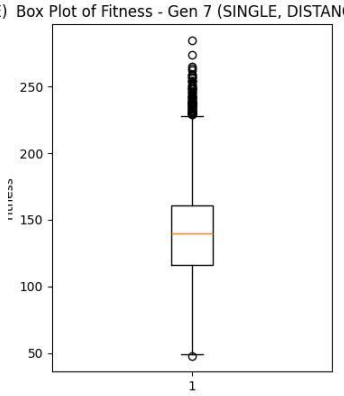


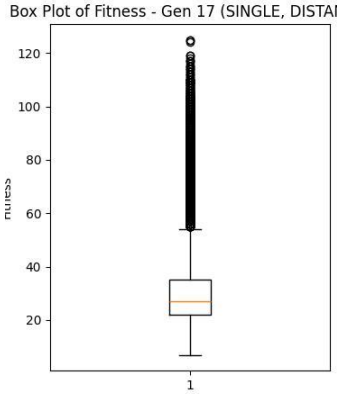
הגרף מציג את התקדמות האלגוריתם לאורך הדורות משלושה נקודות מבט של שלושה מדדים: הפיטנס הטוב ביותר, הפיטנס הגרוע ביותר, והפיטנס הממוצע. ניתן לראות שכולם משתפרים ככל שמספר הדור עולה, במיוחד של הפיטנס הטוב ביותר, דבר המעיד על התכנסות האלגוריתם לפתרון. כמו כן, מהגרף ניתן ללמוד על טווח ערכי הפיטנס לאורך הדורות, כך שאנחנו רואים שהוא מצמצם ככל שמתקדמים בדור, מה שמעיד על ההתייצבות של האוכלוסיה.

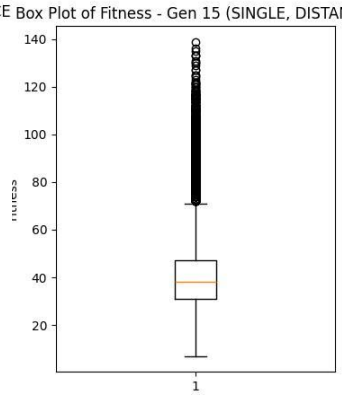


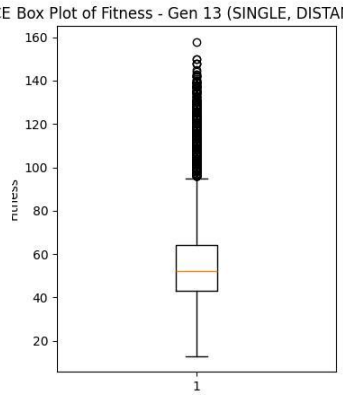

Box Plot of Fitness - Gen 11 (SINGLE, DISTANCE
200
175
150
125
100
75
50
25
1

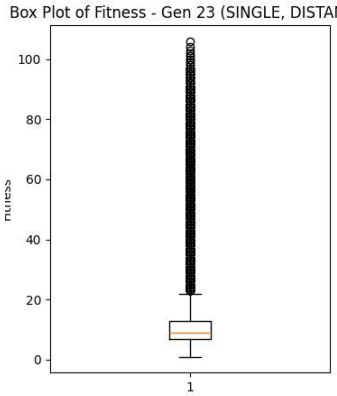

Box Plot of Fitness - Gen 9 (SINGLE, DISTANCE)
225
200
175
150
125
100
75
50
25
1

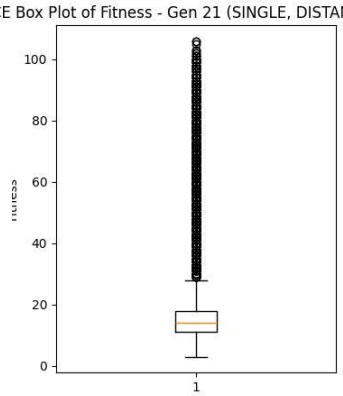

Box Plot of Fitness - Gen 7 (SINGLE, DISTANCE)
250
200
150
100
50
1

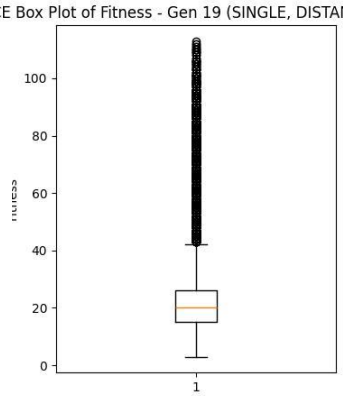

Box Plot of Fitness - Gen 17 (SINGLE, DISTANCE
120
100
80
60
40
20
1

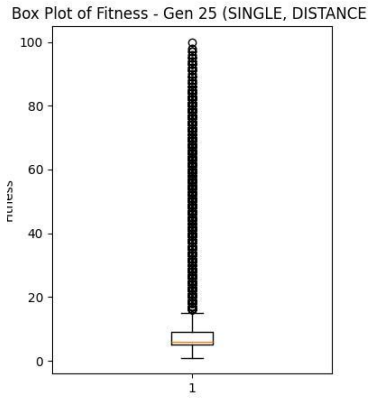

Box Plot of Fitness - Gen 15 (SINGLE, DISTANCE
140
120
100
80
60
40
20
1


Box Plot of Fitness - Gen 13 (SINGLE, DISTANCE
160
140
120
100
80
60
40
20
1


Box Plot of Fitness - Gen 23 (SINGLE, DISTANCE
100
80
60
40
20
0
1


Box Plot of Fitness - Gen 21 (SINGLE, DISTANCE
100
80
60
40
20
0
1


Box Plot of Fitness - Gen 19 (SINGLE, DISTANCE
100
80
60
40
20
0
1



גרף זה מאפשר להבין את פיזור ערכי הפיטניס של האוכלוסייה בכל דור, ומסייע לזהות האם קיימת התכנסות לפתרון, או שהאוכלוסייה "נתקעת" סביב ערכים מסויימים (מידת ה- Exploration לעומת ה- Exploitation). מהגרף ניתן לראות שהערכים מתכנסים בהדרגה לפתרון הטוב ביותר. בנוסף, הגרף משקף את מידת פיזור הערכים, ובגרף רואים שהשונות של הערכים קטנה לאורך הדורות.

סעיף 4

התווספו שלושת הפונקציות המצורפות למטה, כאשר כל אחת מהן מבצעת סוג של שחלוף (Crossover). אחת מהן נקראת לפי אופרטור השיחלוף הנבחר.

```
def single_point_crossover(p1, p2):
    tsize = len(p1)
    spos = random.randint(0, tsize - 1)
    return p1[:spos] + p2[spos:]
```

```
def two_point_crossover(p1, p2):
    tsize = len(p1)
    point1 = random.randint(0, tsize - 1)
    point2 = random.randint(point1, tsize - 1)
    return p1[:point1] + p2[point1:point2] + p1[point2:]
```

```python
def uniform_crossover(p1, p2):
    child = []
    for ch1, ch2 in zip(p1, p2):
        if random.random() < 0.5:
            child.append(ch1)
        else:
            child.append(ch2)
    return "".join(child)
```

<u>סעיף 5</u>

חלקים באלגוריתם האחראים ל- Exploration:

- מוטציות: מאפשרות אקראיות וחשיפה לגנומים חדשים ולא צפויים.
- שיטות שחלוף גבוהות, במיוחד Uniform, אשר יוצרת צאצאים מורכבים ושונים בהשוואה להורים שלהם.
- בחירת הורים ממרחב בגודל חצי האוכלוסייה, שהוא מרחב רחב יחסית, ועל כן מאפשר גיוון ומונע חמדנות.

חלקים באלגוריתם האחראים ל- Exploitation:

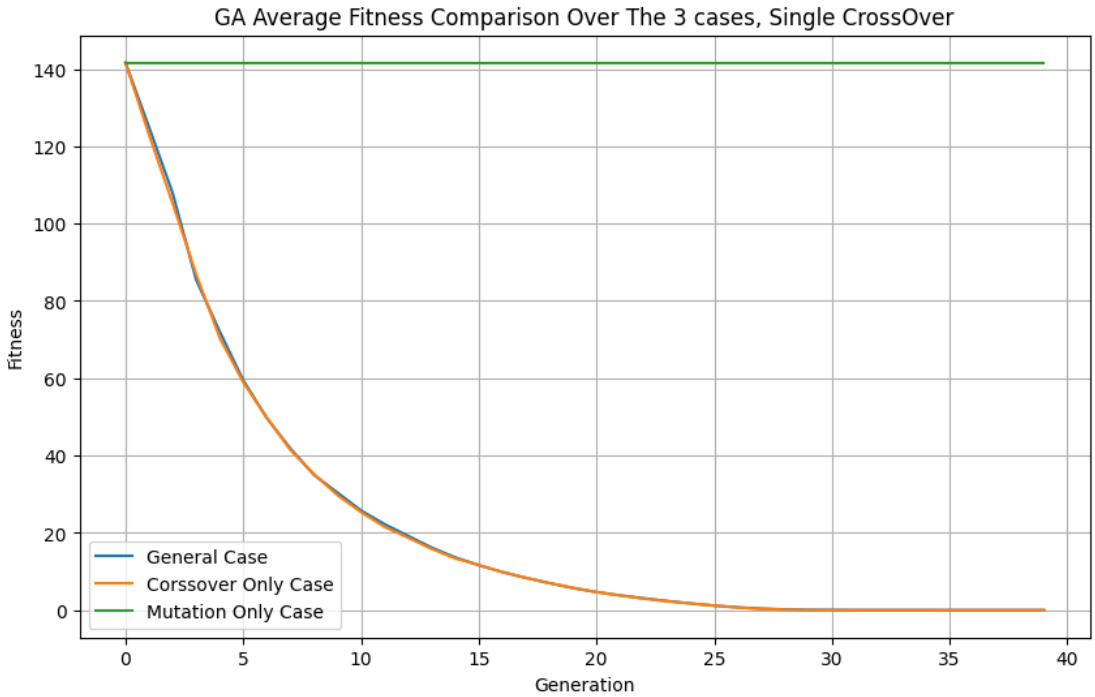
- אליטיזם: בכך שהוא שומר את הפתרונות הטובים ביותר לדור הבא.
- שיטות שחלוף נמוכות, כמו ה- Single-Point, אשר יוצר ילדים הדומים במידה רבה להורים שלהם, או ערבוב שטוח של פרטים מוצלחים.
- בחירת הורים רק מהחצי הטוב של האוכלוסייה: פקטור בעל שני צדדים, אומנם הוזכר כמעודד Exploration, אך הוא מכיל צד שמעודד גם Exploitation כאשר הוא מקדם את החצי המוצלח מהאוכלוסייה על חשבון החצי האחר.

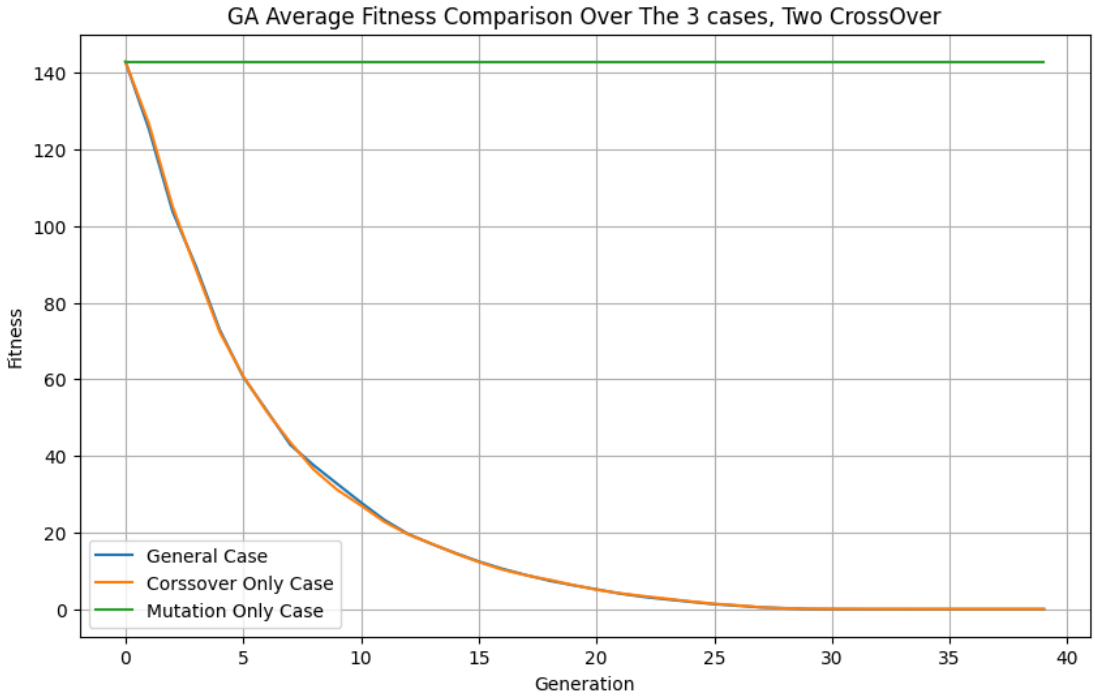
<u>סעיף 6</u>

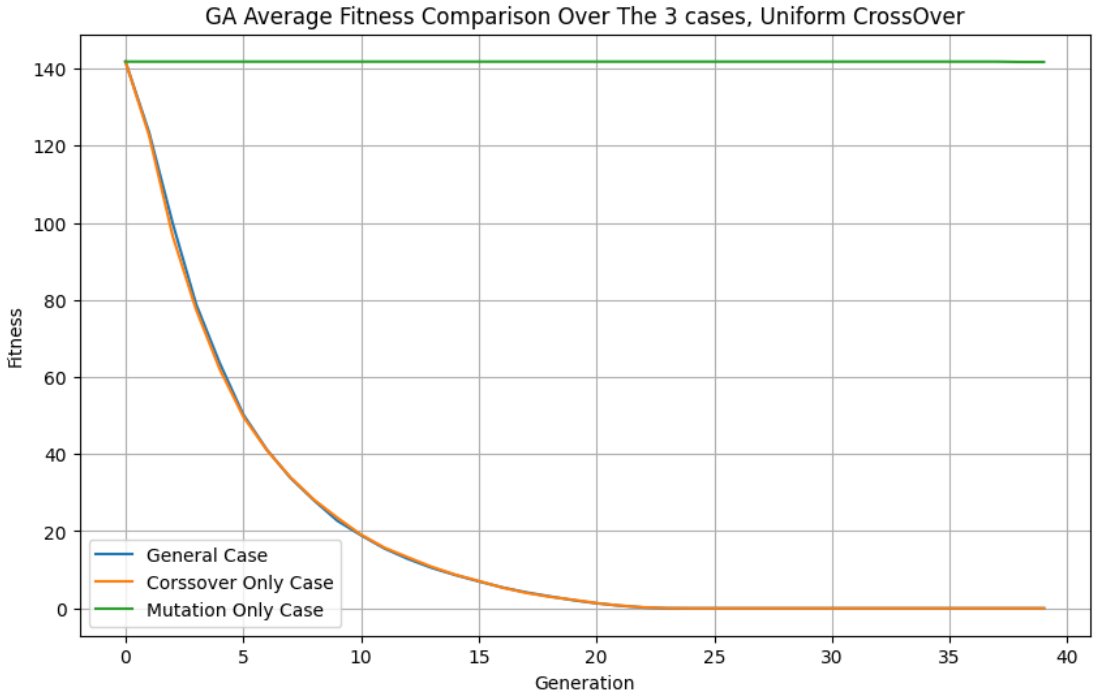
ההשוואה בין שלושת המקרים התבצעה לפי קריטריון הפיטניס הכי טוב לאורך הדורות. הממוצע חושב על פני 100 הרצות שבכל אחת מהן נעשה שימוש באותה אוכלוסייה

התחלתית עבור כל אחד מ- 3 המקרים. האוכלוסייה ההתחלתית הורכבה מפרטים בעלי גינומים שנוצרו באקראי כפי שהדבר מתבצע באלגוריתם הרגיל.

כמו כן, נלקח בחשבון סוג השיחלוף וההשוואה התבצעה בנפרד לכל סוג שיחלוף.







העקומות של שני המקרים, המקרה הכללי (המשלב שיחלוף ומוטציות) והמקרה עם השיחלוף בלבד, כמעט חופפות. מנגד, העקומה של המוטציות בלבד כמעט קבועה לאורך הדורות. מכאן ניתן להסיק שהשיחלוף הוא הגורם המרכזי והמשמעותי להתכנסות, ולמוטציות יש השפעה מזערית.

חקירה נוספת שנעשתה במטרה לבחון את השפעת המוטציות לעומק הראתה שתחת קונפיגורציות שונות במיוחד כאשר מספר האיטרציות גדול יותר (מעל 120), ו\או תנאי העצירה בשל התכנסות מקומית גדול יותר (מעל 50), ו\או הסיכוי למוטציה גבוה יותר (מעל 0.25), העקומה של מקרה המוטציות בלבד כן השתפרה לאורך הדורות, מה שמוביל אותנו למסקנה שההשפעה של המוטציות איטית ונדרשת למרחב זמן ארוך יותר בכדי לבוא לידי ביטוי, כך שההגדרות הנוכחיות שלנו לא מצליחות לתפוס.

סעיף 7

התוספה האפשרות לבחור את הדרך בה הפיטניס מחושב מתוך שתי השיטות: לפי המרחק ממחרוזת המטרה (DISTANCE), או לפי היורסטיקה המוצעת בסעיף (LCS), כאשר האחרונה מתבצעת באמצעות הפונקציה המצורפת למטה, אשר משתמשת באלגוריתם תכנון דינמי על מנת למצוא את הרצף המשותף הארוך ביותר בין שתי המחרוזות. בנוסף, נעשה שימוש ב-

Backtracking על מנת לבדוק אם מיקומי האותיות ברצף הזה דומות בשתי המחרוזות, ובונוס מוענק בהתאם.

ערך הבונוס הוא 4.0, והוא נבחר מתוך טווח ערכים 0.0 עד 10.0, בקפיצות של 0.5, על בסיס שני קריטריונים המצוינים לפי סדר העדיפות: (1) אחוז גבוה יותר של ריצות שהתכנסו לפתרון אופטימלי, (2) התכנסות מהירה יותר (מספר דורות נמוך יותר עד ההתכנסות). הבדיקה בוצעה על פני 100 אוכלוסיות שונות שאותחלו באופן אקראי.

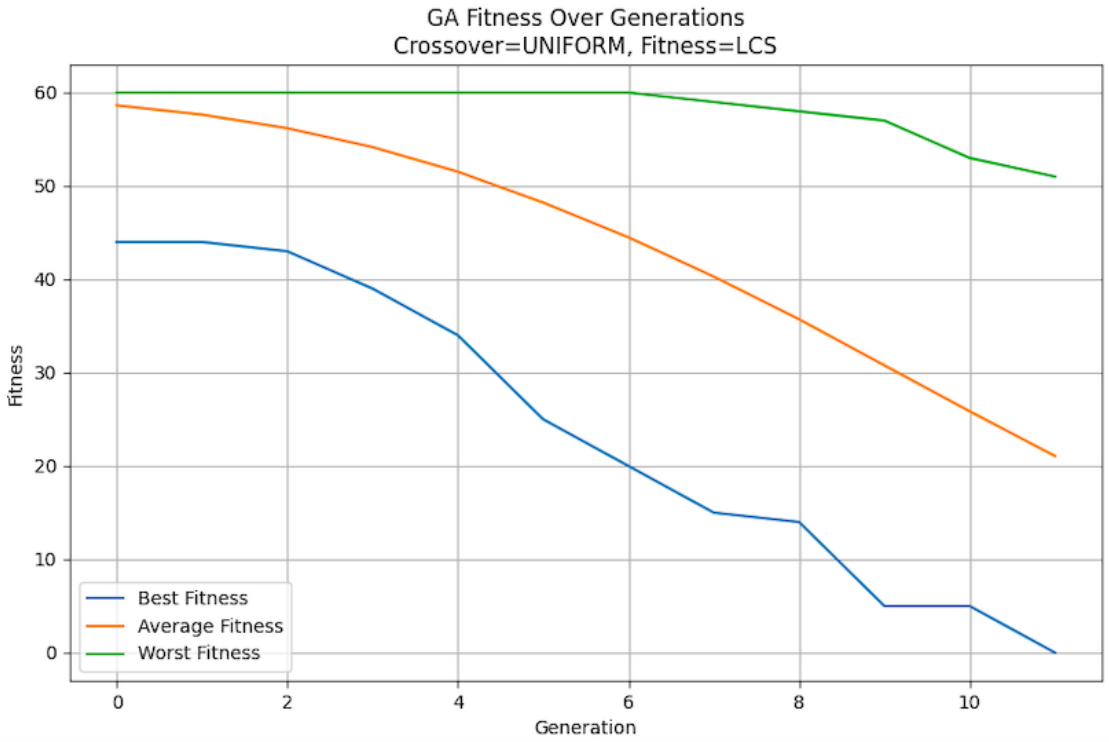
```python
def fitness_by_lcs(self, a, b):
    m = len(a)
    n = len(b)
    L = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if a[i - 1] == b[j - 1]:
                L[i][j] = L[i - 1][j - 1] + 1
            else:
                L[i][j] = max(L[i - 1][j], L[i][j - 1])
    lcs_length = L[m][n]

    correct_chars_count = 0
    bonus = 4
    while m > 0 and n > 0:
        if a[m - 1] == b[n - 1]:
            m -= 1
            n -= 1
            if m == n:
                correct_chars_count += 1
        elif L[m - 1][n] > L[m][n - 1]:
            m -= 1
        else:
            n -= 1
    max_possible = (bonus+1) * len(b)

    return max_possible - (lcs_length + bonus *
                           correct_chars_count)
```

a. היוריסטיקה החדשה מקדמת פתרונות המכילים רצף נכון שהאותיות שלו במקומות הנכונים. בגלל שהאלגוריתם בוחר הורים מהפריטים הכי טובים, שהם לפי הגדרת היוריסטיקה הזו כאלה המכילים רצף אותיות הממוקמות נכון, קל יותר לאלגוריתם כעת ליצור משני הורים גינום שמכיל רצף משותף ארוך יותר ובעל מספר גדול יותר של אותיות הממוקמות נכון.

b. היוריסטיקה החדשה עדיפה על המקורית באופן מובהק. השיפור נצפה גם באחוז הפעמים בהם יש התכנסות לפתרון הנכון וגם בממוצע מספר הדורות הצריך עד להתכנסות. באשר לסיכוי ההתכנסות אין שיפור גדול מפני שתחת הפרמטרים האופטימליים האלגוריתם התכנס באחוז גבוה עוד לפני מימוש היוריסטיקה, אולם כן נצפה שיפור משמעותי במספר הדורות שעמד על ממוצע של 10.41 לעומת הטווח של 25-29 בהיוריסטיקה הקודמת.

תצפית נוספת שראויה לציין היא שמידת השיפור בכל אחד מבין 3 המדדים היא בסדר הזה: הפיטניס הטוב ביותר, הפיטניס הממוצע ואז הפיטנס הגרוע ביותר, שלא משתפר בהרבה. התנהגות זו הפוכה מההתנהגות של שלושת המדדים כאשר השתמשנו ביורסטיקת ה- Distance. כתוצאה מכך גם טווח הפיטניס מתנהג באופן הפוך והוא גדל לאורך הדורות. הדבר נובע כנראה מאופן התנהגות השיטה, שבאותה מידה שבה היא מסוגלת לייצור מחרוזת בעלת רצף ארוך יותר ומדויק יותר משני ההורים, היא גם עלולה לייצור מחרוזת גרועה מהם שמקלקלת את הרצפים ומתרחקת מהפתרון.

להלן התוצאות של 3 המדדים שהתקבל מאחת ההרצות:



שתי השיטות מומשו בדרך שהיא לא ספציפית לדרך הנוכחית בה משתמשים ל- Parent selection, אלא בדרך שתאפשר את מדידת לחץ הבחירה לכל מנגנון Parent selection שיבחר בהמשך.

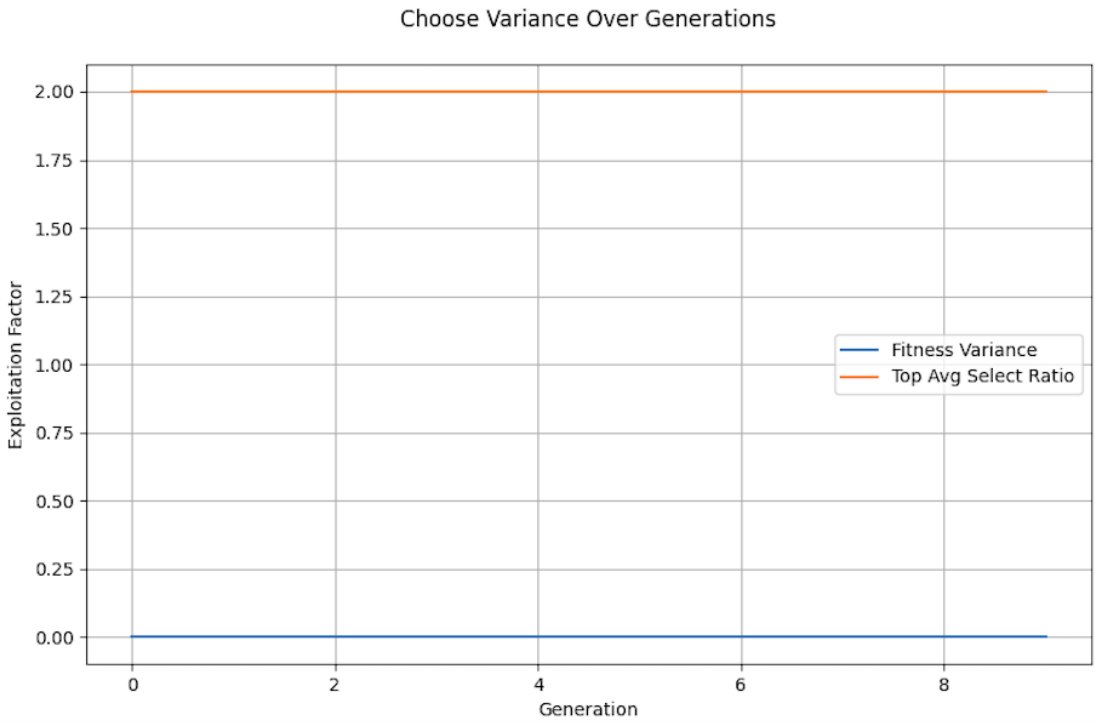
a. לצורך מימוש שיטת Fitness Variance השתמשנו בכל דור במערך מונים, המכיל מונה אחד לכל פרט, שסופר את מספר הפעמים שבהם הפרט נבחר כהורה להליך הריבוי. עם סיום הדור, המערך נשלח לפונקציה המצורפת למטה, אשר מחשבת את השונות של ערכי המונים. בתום ריצת האלגוריתם הערכים של כלל הדורות מוצגים ע"י גרף.

```
def fitness_variance(select_count, population_size):
    choose_prob = [count / population_size
                        for count in select_count]
    avg_choose_prob = sum(choose_prob) / population_size
    variance = sum((p - avg_choose_prob) ** 2 for p
                    in choose_prob) / population_size
    return variance
```

b. למימוש שיטת Top-Average Selection Probability Ratio השתמשנו באותו מונה מהסעיף הקודם, אשר נשלח בסיום הדור לפונקציה המצורפת למטה. הפונקציה בתורה מחשבת את היחס בין ההסתברות שההורה נבחר מהחצי הכי טוב של האוכלוסייה להסתברות הממוצעת לבחירת פרט כלשהו מהאוכלוסייה. בסוף ריצת האלגוריתם הערכים שהתקבלו מכלל הדורות מוצגים ע"י אותו גרף מהסעיף הקודם.

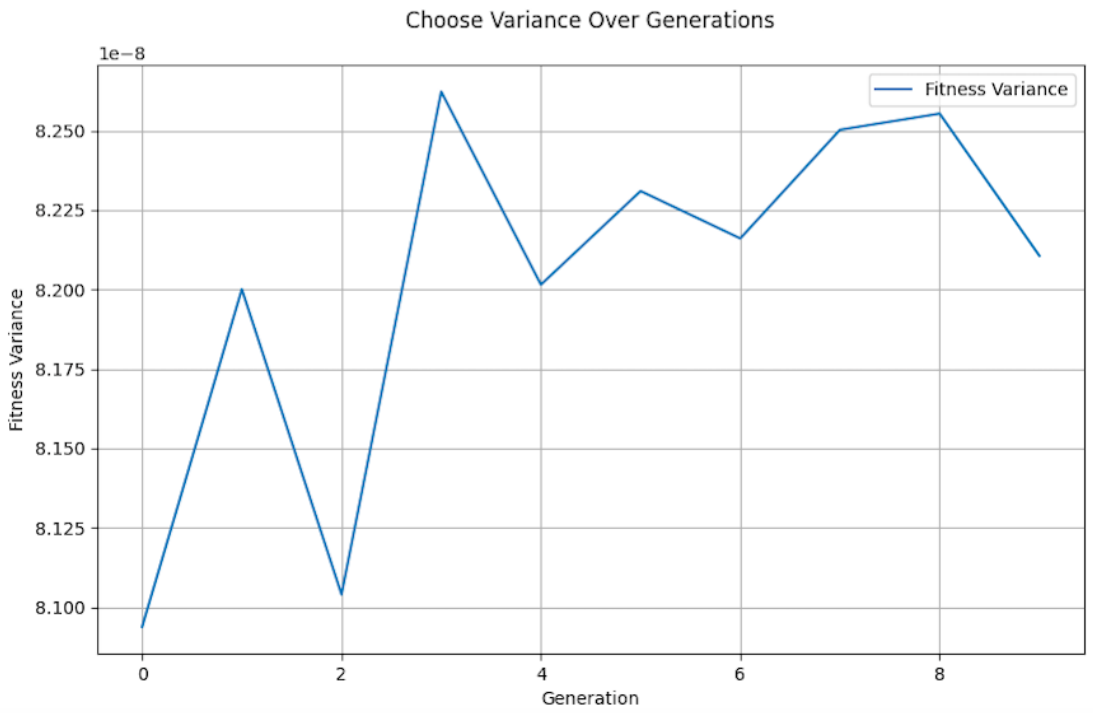
```
def top_avg_select_ratio(select_count):
    top_half = select_count[:len(select_count)//2]
    top_avg = sum(top_half) / len(top_half)
    all_avg = sum(select_count) / len(select_count)
    return top_avg / all_avg if all_avg != 0 else 0
```

להלן גרף המציג את ערכי המדדים שהתקבלו מאחת ההרצות ע"י שתי השיטות:



מכיוון ששיטת בחירת ההורים הנוכחית בוחרת את ההורים באופן אקראי מבין החצי הכי טוב של האוכלוסייה, הערכים אינם משתנים כתלות בדור או בפיטניס, אלא תלויים בגודל האוכלוסייה ובפקטור ה- Elitism.

למרות שבאופן תאורטי הערכים של ה- Fitness Variance אמורים להישאר קבועים לאורך הדורות, בפועל בשל הטיית הדגימה הם לא כאלה, אולם ערכיהם קרובים מאוד. הגרף למעלה לא מראה זאת באופן ברור בשל ה- Scaling שמושפע מערכי המדד האחר. כך זה נראה בגרף שה- Scaling שלו מותאם לסקלה של ערכי ה- Fitness Variance:



מדד ה- Fitness Variance מבטא את רמת הפיזור של סיכויי הבחירה באוכלוסייה. במלים אחרות, עד כמה סיכויי הבחירה של הפרטים באוכלוסייה שונים זה מזה. שונות גדולה יותר מצביעה על לחץ בחירה גדול יותר. זה קורה כאשר חלק קטן מהאוכלוסייה זוכה להעדפה על פני השאר. מנגד ככל שההסתברות להיבחר דומה יותר, השונות נהיית נמוכה, והחלץ בחירה נהיה חלש יותר.

מדד ה- Top-Average Selection Probability Ratio מבטא את מידת ההעדפה לקבוצת הפרטים הטובים ביותר באוכלוסייה. ערך גדול יותר מצביע על לחץ בחירה גדול יותר, וזה קורה כאשר האלגוריתם נותן משקל גדול יותר לפרטים הטובים בתהליך הריבוי.

סעיף 9

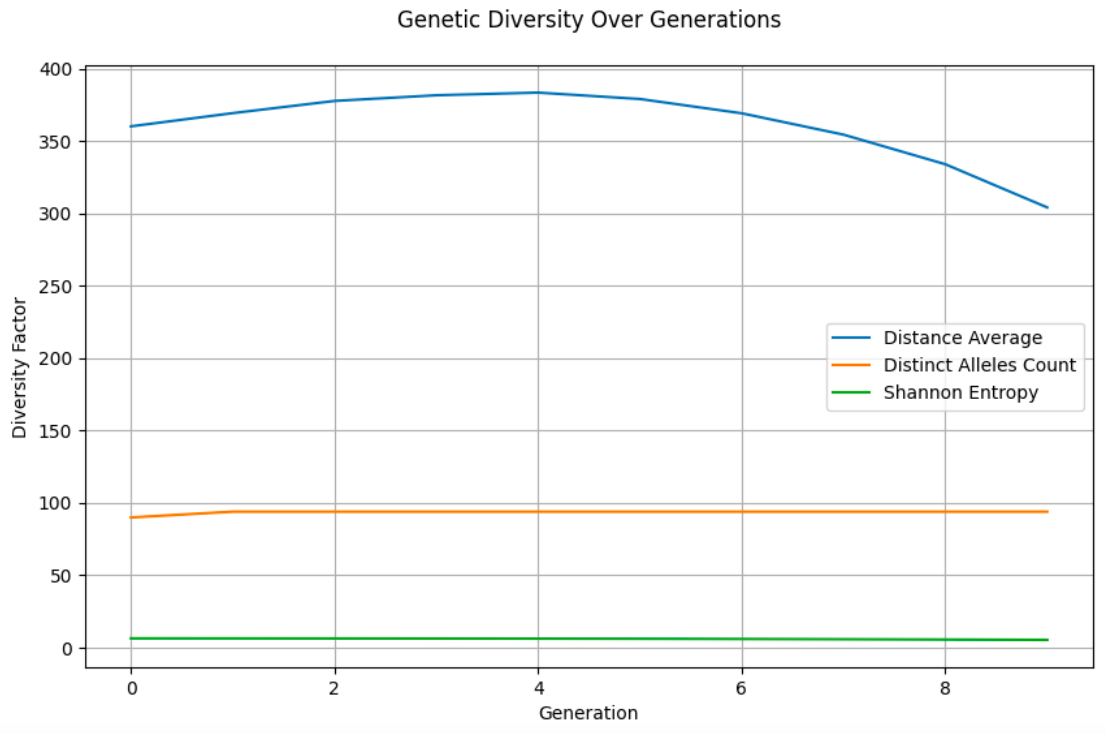
מימוש כל אחת מהשיטות התבצע בפונקציה נפרדת, שלושת הפונקציות צורפו למטה.
הפונקציות נקראות בתום כל דור, והערכים של כלל הדורות מוצגות באמצעות גרף בסיום ריצת האלגוריתם.

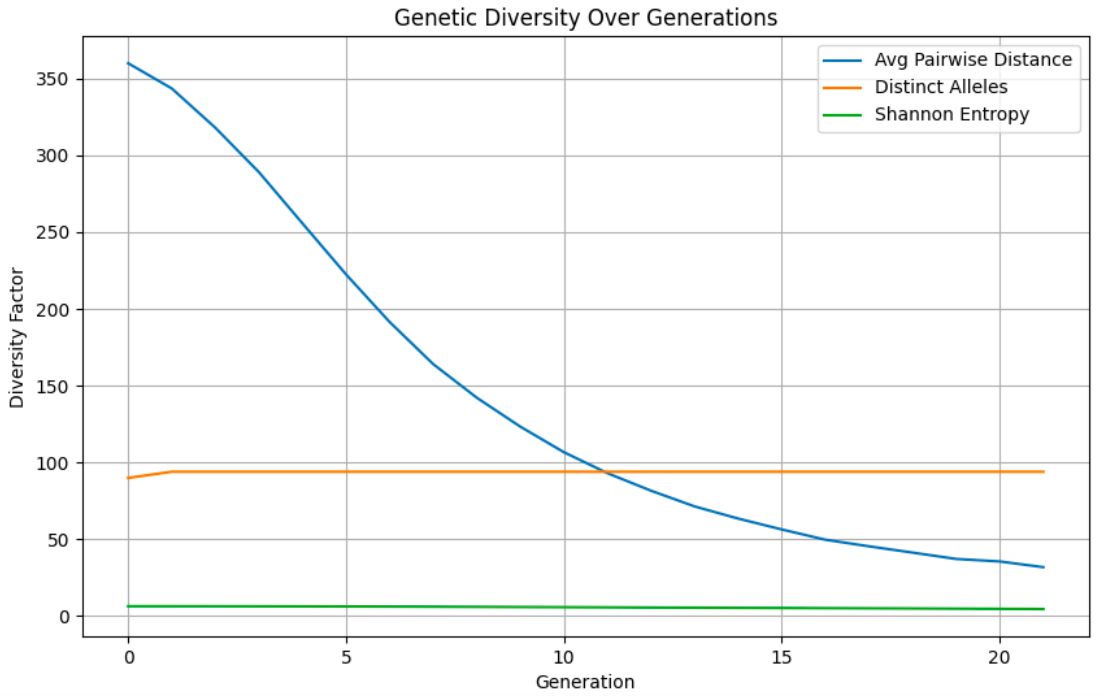
```python
def distance_average(individuals):
    if len(individuals) < 2:
        return 0.0

    genome_length = len(individuals[0].genome)
    total_distance = 0
    n = len(individuals)

    for pos in range(genome_length):
        freq = {}
        for ind in individuals:
            ch = ind.genome[pos]
            if ch in freq:
                freq[ch] += 1
            else:
                freq[ch] = 1

        chars = list(freq.items())
        for i in range(len(chars)):
            char_i, count_i = chars[i]
            for j in range(i + 1, len(chars)):
                char_j, count_j = chars[j]
                diff = abs(ord(char_i) - ord(char_j))
                total_distance += count_i * count_j * diff

    total_pairs = n * (n - 1) // 2
    return total_distance / total_pairs
```

```python
def distinct_alleles(individuals):
    all_chars = set(ch for ind in individuals
                        for ch in ind.genome)
    return len(all_chars)
```

```python
def shannon_entropy(individuals):
    char_counts = {}
    total_chars = 0
    for ind in individuals:
        for ch in ind.genome:
            char_counts[ch] = char_counts.get(ch, 0) + 1
            total_chars += 1
    entropy = -sum((count / total_chars) * math.log2(count / total_chars)
                   for count in char_counts.values() if count > 0)
    return entropy
```

כמו כן, מצורפות תוצאות שלושת המדדים עבור שתי ריצות, הראשונה השתמשה בהיוריסטיקת ה- LCS, והשנייה בהיוריסטיקת ה- DISTANCE.





מדד "המרחקים בין הפרטים באוכלוסייה", כשמו, מודד עד כמה הפרטים באוכלוסייה רחוקים זה מזה מבחינת ערכי ASCII המרכיבים את הגנום שלהם, ולכן ערך גבוה יותר משקף שונות גדולה יותר בין הפרטים, כלומר גיוון גדול יותר בין הפתרונות. מדד "מספר האללים השונים" סופר את כמות התווים הייחודיים המופיעים באוכלוסייה ללא קשר למיקום או התדירות שלהם, ומבטא את הגיוון באוכלוסייה. מדד "האנטרופיה לפי שנון" מודד את מידת האקראיות בחלוקת האללים באוכלוסייה, ומודד את הגיוון כמו קודמו, אך גם לוקח בחשבון את הכמויות (ההתפלגויות של הערכים). ערך גבוה בו משקף כמות גדולה של תווים המתפלגים באופן מאוזן.

ההבדלים בין שלושת המדדים הוא באופן שבו הם מגדירים את הגוון הגנטי, כאשר הראשון עושה את זה לפי המרחקים בין המחרוזות ללא קשר לתוכן שלהן. השני מדגיש יותר את הגיוון בתוכן ולא בהבדלים אינדיבידואליים, ואילו השלישי לוקח בחשבון גם את התדירויות וכמות הופעות כל אחד מאיברי התוכן.

שני המדדים "מספר האללים" ו- "האנטרופיה לפי שנון" נשארו כמעט קבועים לאורך הדורות, ודומים בין ריצות שתי היוריסטיקות. מספר האללים היה שווה ל- 90 באופן קבוע החל מהדור השני, שזה מספר ערכי ה- ASCII השונים שאות יכול לקבל. הדבר נובע ככל הנראה מהמוטציות, אך יותר בשל שיטת בחירת ההורים הרחבה יחסית (באופן אקראי מהחצי הטוב ביותר) המשמרת הרבה מהערכים הלא רלוונטיים. ערכי מדד "אנטרופיית שנון" שגם הוא נשאר קבוע מעיד על תדירות הופעה לא משתנה מהותית של התווים לאורך הדורות. לעומת זאת, מדד "המרחק בין הפריטים" הראה שוני באופן מובהק, גם לאורך הדורות, וגם באופן ההתנהגות בין שתי הריצות. בהיוריסטיקת ה- LCS ערך המדד ירד באופן עקבי החל מהדור החמישי, בעוד שבהיוריסטיקת ה- DISTANCE הירידה החלה כבר מהדור הראשון, דבר המעיד על התכנסות לפרטים טובים יותר. הירידה הגדולה במקרה ה- DISTANCE לעומת ה- LCS תומכת בהבחנה שצוינה מקודם אודות יכולת ה- DISTANCE לשפר גם את הפתרונות הגרועים (בין היתר הפרט הכי גרוע) בצורה טובה יותר לעומת ה- LCS.

<u>סעיף 10</u>

ארבעת שיטות בחירת ההורים בנוסף לשיטת השרידות מומשו באמצעות הפונקציות המצורפות להלן:

השיטה שמבצעת בחירת הורים ע"י RWS:

```python
def rws_selection(individuals):

    max_fitness = max(ind.fitness for ind in individuals)
    scaled_fitnesses = linear_scaling(individuals,
                                      -1, max_fitness)

    total_scaled = sum(scaled_fitnesses)
    selection_probs = [fit / total_scaled
                           for fit in scaled_fitnesses]

    if total_scaled == 0:
        random_choice = random.randint(0, len(individuals) - 1)
        return random_choice, individuals[random_choice].genome

    cumulative_probs = []
    cumsum = 0
    for prob in selection_probs:
        cumsum += prob
        cumulative_probs.append(cumsum)

    pick = random.random()
    for i, prob in enumerate(cumulative_probs):
        if pick < prob:
            return i, individuals[i].genome
    return len(individuals) - 1, individuals[-1].genome
```

השיטה שמבצעת בחירת הורים ע"י SUS:

```python
def sus_selection(individuals, num_parents):

    max_fitness = max(ind.fitness for ind in individuals)
    scaled_fitness = linear_scaling(individuals,
                                    -1, max_fitness)

    total_fitness = sum(scaled_fitness)
    selection_probs = [fit / total_fitness
                       for fit in scaled_fitness]

    cumulative_probs = []
    cumsum = 0
    for prob in selection_probs:
        cumsum += prob
        cumulative_probs.append(cumsum)

    rand = random.random() / num_parents
    parents = []
    for i in range(num_parents):
        target = rand + i / num_parents
        for j, prob in enumerate(cumulative_probs):
            if target < prob:
                parents.append((j, individuals[j].genome))
                break
    return parents
```

השיטה שמבצעת בחירת הורים ע"י Deterministic Tournament:

```python
def tournament_selection_deter(individuals):
    tournament = random.sample(range(len(individuals)), TOURNAMENT_K)
    tournament.sort(key=lambda ind: individuals[ind].fitness)
    return tournament[0], individuals[tournament[0]].genome
```

השיטה שמבצעת בחירת הורים ע"י Non-Deterministic Tournament:

```python
def tournament_selection_stoch(individuals):
    tournament = random.sample(range(len(individuals)), TOURNAMENT_K)
    tournament.sort(key=lambda ind: individuals[ind].fitness)
    for i in range(TOURNAMENT_K):
        if random.random() < TOURNAMENT_P:
            return tournament[i], individuals[tournament[i]].genome
    return tournament[-1], individuals[tournament[-1]].genome
```

השיטה שמבצעת Linear Scaling:

```python
def linear_scaling(individuals, a,b):
    scaled_fitness = [a * ind.fitness + b
                        for ind in individuals]
    return scaled_fitness
```

השיטה שמבצעת Fitness Ranking:

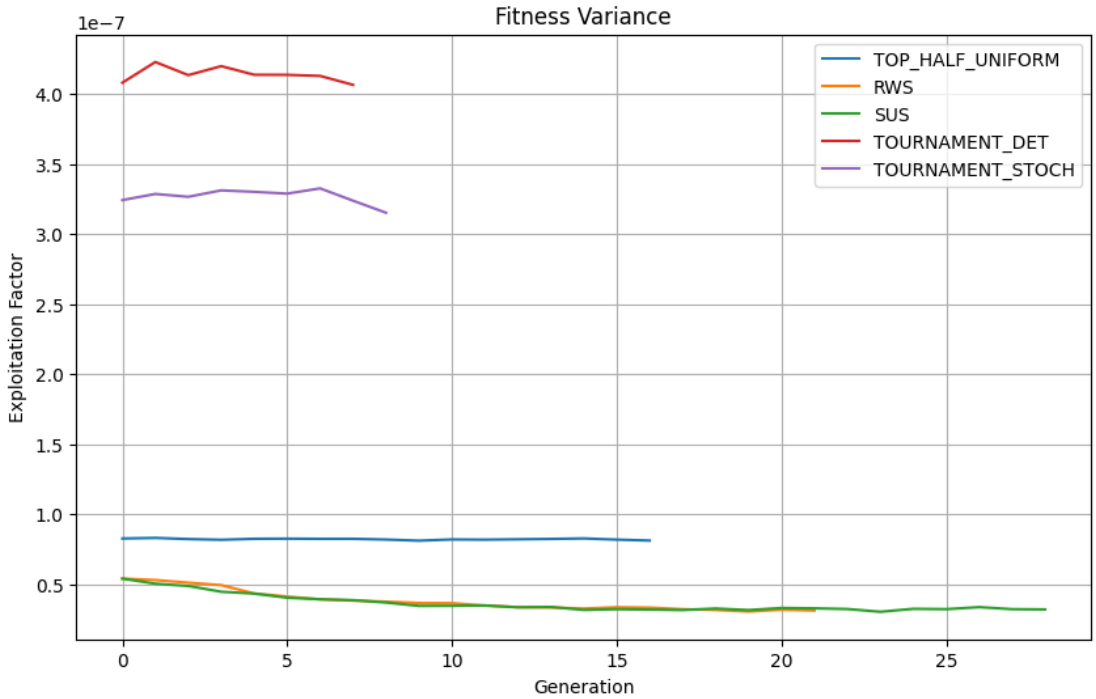
```python
def fitness_ranking(individuals, reverse=False):
    sorted_inds = sorted(individuals, key=lambda ind:
                         ind.fitness, reverse=reverse)
    for rank, ind in enumerate(sorted_inds):
        ind.rank = rank + 1
    return sorted_inds
```

השיטה שמבצעת Aging:

```python
def aging(population):
    for ind in population.individuals:
        ind.age += 1
    population.individuals = [ind for ind in population.individuals
                                if ind.age < AGE_LIMIT]
```

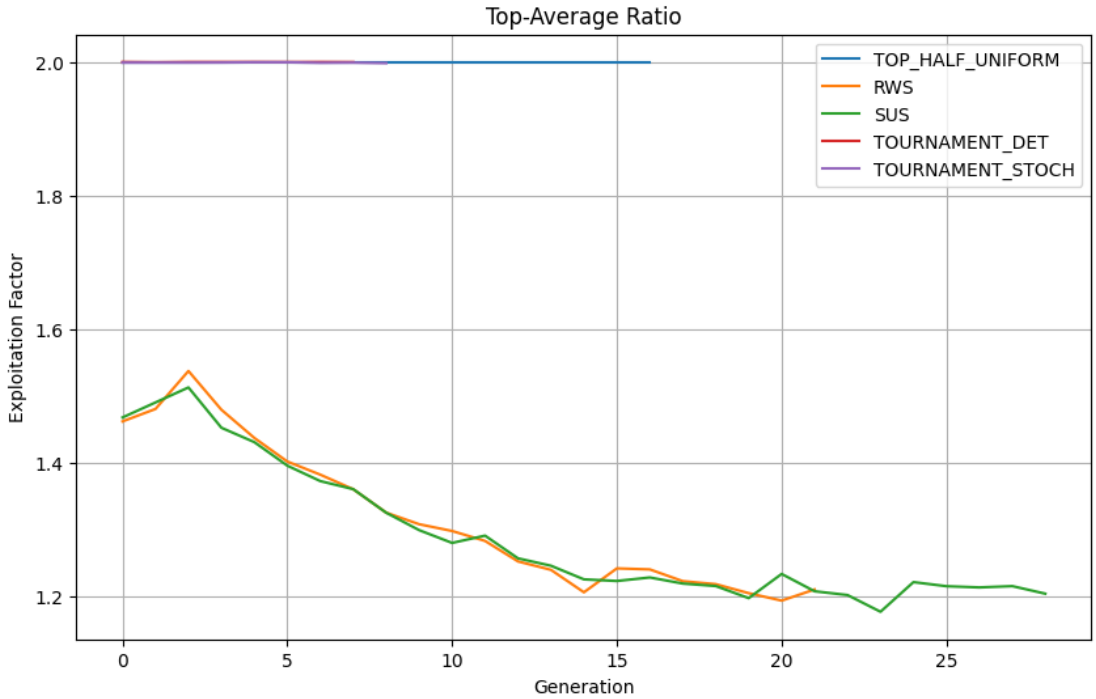
הערכים של הפרמטרים P, K, AGE_LIMIT נבחרו לפי הקריטריונים של אחוז התכנסות גלובלי ומספר מינימלי של דורות. הקריטריונים הושגו ע"י מיצוע תוצאות של 100 הרצות. הערך של K נבחר בהינתן ערך ה- P האופטימלי שנמצא קודם.

הגרפים הבאים מראים את השתנות כל אחד ממדדי הלחץ הגנטי והגיוון לאורך הדורות כתלות בשיטת בחירת ההורות:

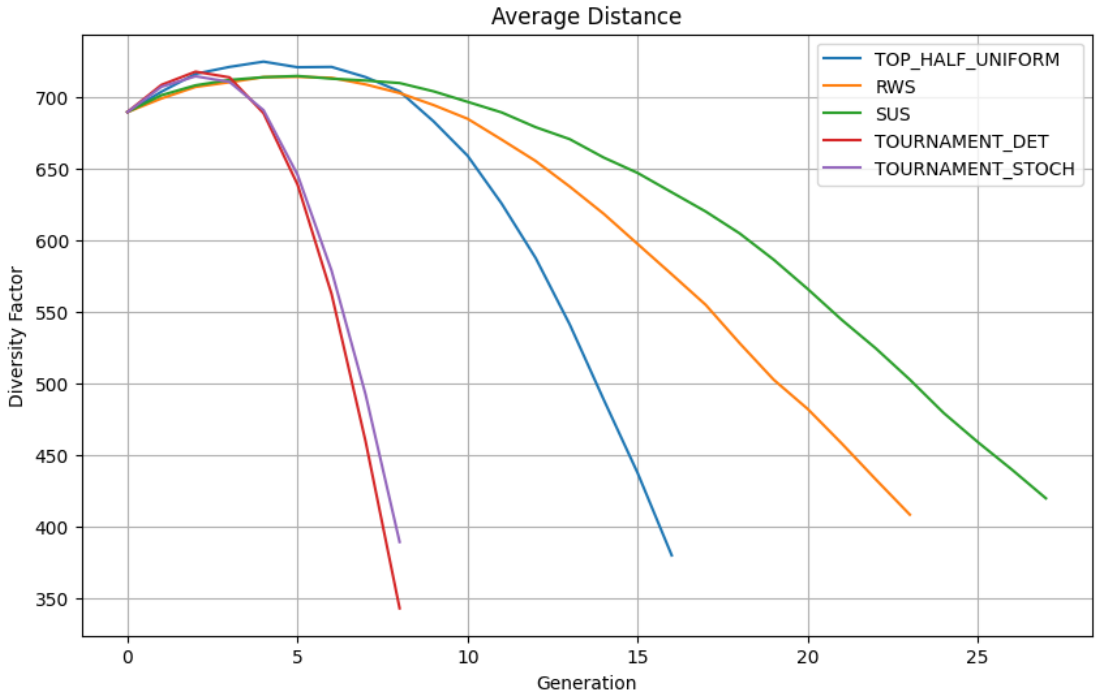


מדד **Fitness Variance**: שיטת ה- "טורניר דטרמיניסטי" יוצרת את הלחץ החזק ביותר, כפי שניתן לראות מהשונות הגבוהה, וזאת תוצאה של בחירה תמידית בפרטים הטובים ביותר, מה שמוביל להתכנסות מהירה אך גם לסיכון להיתקע במינימום מקומי. שיטת ה- "טורניר לא דטרמיניסטי" שומרת על לחץ גנטי גבוה גם כן, אך באופן מעט מרוכך, כנראה בשל אלמנט הרנדומליות שבה, שמאפשר לפרטים פחות טובים להיבחר מדי פעם. מנגד, שיטות RWS ו- SUS מציגות שונות נמוכה יותר, כלומר לחץ גנטי חלש יותר משל השיטה ההתחלתית (שבחרנו לקרוא לה "Top Half Uniform"), המאפשר שמירה על גיוון באוכלוסייה לאורך זמן ועידוד של Exploration על פני ניצול Exploitation מהיר של פתרונות קיימים.

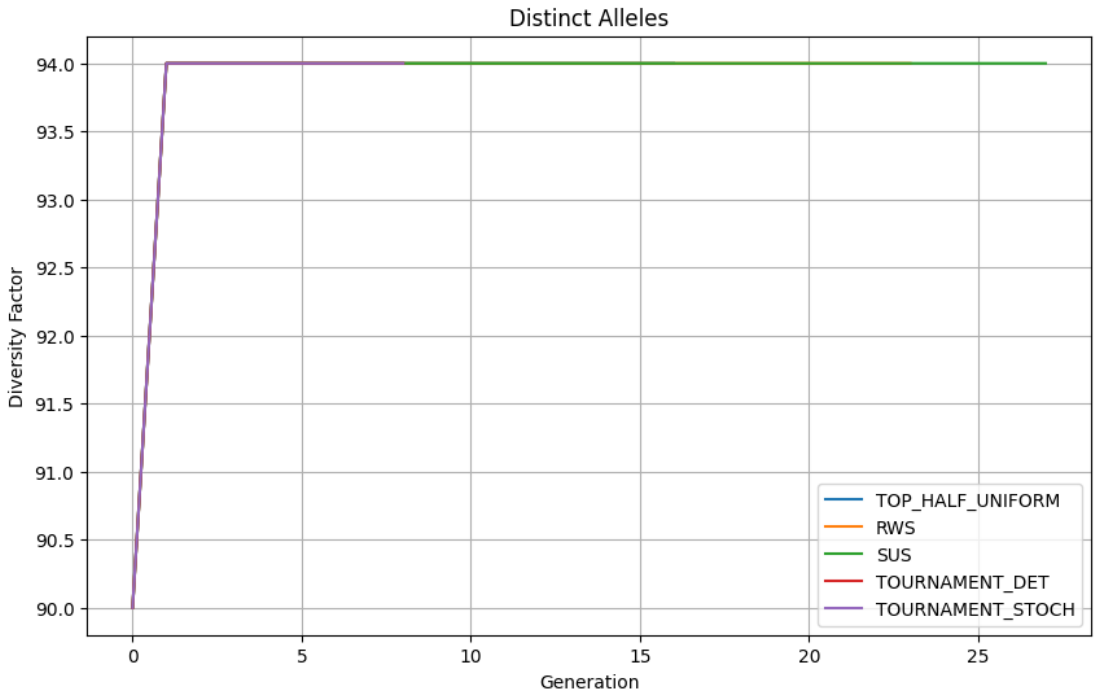
בנוסף, ניתן להבחין בכך שבכל ארבעת השיטות יש נטייה מסוימת לירידה של השונות עם התקדמות הדורות. במיוחד בשיטות RWS ו-SUS.



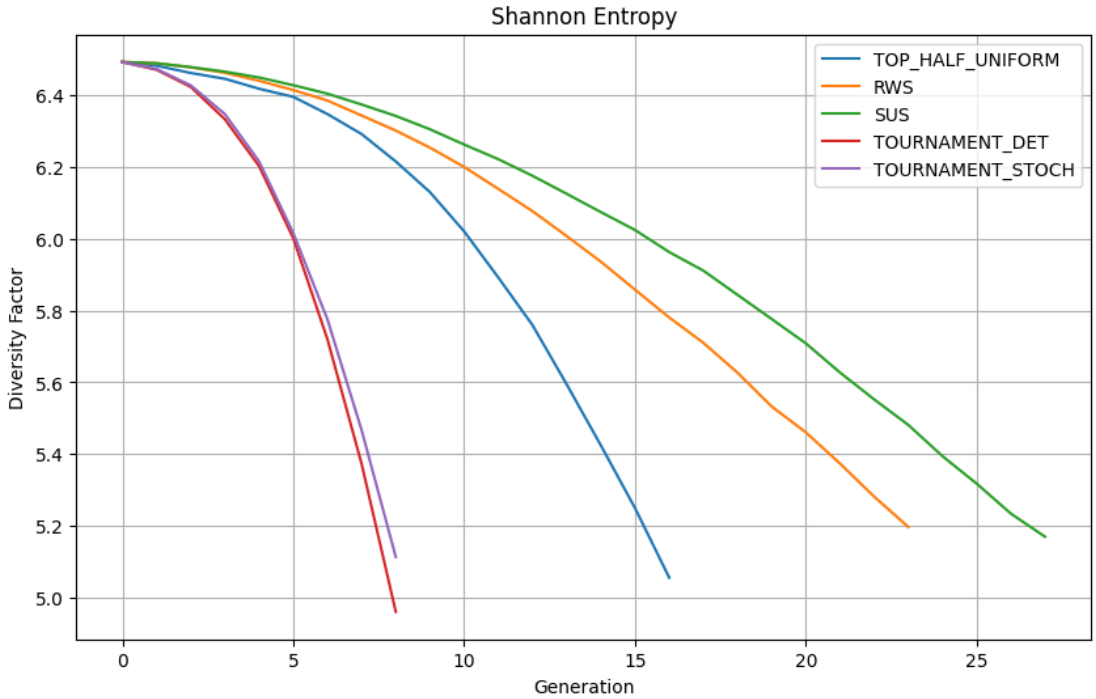
מדד **Top-Average Selection Probability Ratio**: גם כאן השיטות "טורניר דטרמיניסטי" ו-"טורניר לא דטרמיניסטי" מראות לחץ גנטי גבוה, והוא קבוע גם כן לאורך הדורות. דבר זה הגיוני לאור כך שהסיכוי ששיטת טורניר תבחר פרט מהחצי התחתון הוא רק אם כל K הפרטים שהוגרלו הם מהחצי התחתון (למרות שזה לא הכי מדויק בנוגע לטורניר הלא דטרמינסטי). לעומת זאת, בשיטות RWS ו-SUS לחץ הבחירה נמוך מאשר שיטות הטורניר, והוא חווה ירידה הדרגתית ביחס הזה עם הזמן.



מדד **המרחק בין הפרטים באוכלוסייה**: ניתן לראות מגמת ירידה בגיוון עם התקדמות הדורות בכל השיטות. עם זאת, שיטות הטורניר מציגות ירידה חדה מאוד כבר מהדור החמישי – מה שמעיד על התכנסות מהירה מאוד. מנגד, שיטות RWS ו-SUS מצליחות לשמר גיוון גבוה יותר למשך זמן ארוך יותר, מה שמעיד על חקר רחב יותר של המרחב הגנטי.



מדד **מספר האללים השונים באוכלוסיה**: המדד מראה יציבות גבוה בכל השיטות, כאשר כולן מגיעות מהר ל- 94 אללים, שזה גודל טווח האללים, והוא נשמר לאורך כל הדורות.



מדד האנטרופיה לפי שנון: התוצאות דומות מאד לאלה שהתקבלו במדד "המרחק בין הפרטים", כאשר שני מדדי הטורניר יורדים מהר, מה שאומר שהאוכלוסייה מתכנסת לערכים שחוזרים על עצמם במהירות. לעומתם, RWS ו-SUS מציגות ירידה הדרגתית ושמירה על גיוון לאורך יותר זמן.

סעיף 11

בסעיף זה התבקשנו לפתור את בעיית ה – Bin Packing. הרעיון הכללי בבעיה הוא שיש לנו כמות מסוימת של איברים עם גדלים שונים, והמטרה היא להכניס את כל האיברים למספר הקטן ביותר של פחים (Bins) כאשר לפח יש קיבולת מקסימלית. כדי למצוא פתרון טוב לבעיה השתמשנו באלגוריתם שבנוי על עקרונות של אבולוציה - כלומר יצירת פתרונות אקראיים, בדיקה שלהם, שיפור לאורך דורות, ובחירה של הפתרונות הכי טובים.

איך האלגוריתם עובד ? בתחילת האלגוריתם אנחנו יוצרים אוכלוסייה של הרבה פתרונות אקראיים. כל פתרון הוא דרך מסוימת לסדר את האיברים בפחים. השתמשנו בשיטת Best Fit ליצירת הפחים, כלומר האיבר נכנס לפח שהכי מתמלא, ואם אין פח שיכול להכיל אותו נפתח פח חדש. אחרי שבנינו את כל הפתרונות, אנחנו מחשבים לכל אחד מהם את ערך הפיטניס, שזה פשוט כמות הפחים שהוא השתמש בהם פחות מספר הפחים שהאלגוריתם האופטימלי השתמש בהם. כמה שפחות פחים = פתרון טוב יותר. בכל דור אנחנו שומרים את הפתרונות הכי טובים שמצאנו (Elitism), ויוצרים פתרונות חדשים ע"י לשמור עבור כל

פרט את ה- 50% מהפחים הכי מלאים, ולערבב מחדש את האיברים של 50% הפחים הפחות מלאים. האיברים המעורבבים בודקים בין היתר אם יש מקום בפחים הכי מלאים אשר נשמרו. אנחנו ממשיכים ככה בלולאה, דור אחרי דור, עד להתכנסות אופטימלית, התכנסות מקומית או חריגה מהזמן שהוקצה לאלגוריתם.

השתמשנו באותם מבני נתונים של הבעיה המקורית עבור הפרטים והאוכלוסייה, עם שינויים והוספות שנועדו להתאים אותם לבעיה.

להלן **הפונקציות החדשות שהתווספו**:

```
def bin_packing_init_population(self, initial_genome):
    population = []
    for _ in range(self.size):
        individual = Individual(initial_genome)
        genome = individual.best_fit(initial_genome,
                                     self.bin_capacity)
        individual.genome = genome
        population.append(individual)
    return population
```

```
def best_fit(self, initial_genome, bin_capacity):
    random.shuffle(initial_genome)
    genome = []
    for item in [item for bin in initial_genome
                 for item in bin]:
        best_bin_index = -1
        min_space_left = bin_capacity+ 1
        for i, bin in enumerate(genome):
            space_left = bin_capacity - sum(bin)
            if space_left >= item and space_left - item < min_space_left:
                best_bin_index = i
                min_space_left = space_left - item
        if best_bin_index == -1:
            genome.append([item])
        else:
            genome[best_bin_index].append(item)

    return genome
```

שתי הפונקציות ביחד יוצרת אוכלוסייה התחלתית. הן עושות זאת ע"י קבלת גינום התחלתי שהוא גינום המכיל איבר אחד בכל פח, מחלצות את איבריו, מערבבות אותם, ומסדרות אותם מחדש בתוך פחים באמצעות שיטת Best Fit שהוסברה מקודם.

```python
def first_fit(self, initial_genome, bin_capacity):
    random.shuffle(initial_genome)
    genome = []
    for item in [item for bin in
                 initial_genome for item in bin]:
        placed = False
        for bin in genome:
            if sum(bin) + item <= bin_capacity:
                bin.append(item)
                placed = True
                break
        if not placed:
            genome.append([item])
    return genome
```

הפונקציה עושה את אותה הפונקציונליות של Best Fit. אך במקום לבחור עבור הפרט את הפח שהכי מתמלא, היא בוחרת את הפח הראשון שהוא יכול להיכנס אליו. ואם אין פח כזה, היא יוצרת אחד חדש. האלגוריתם לא משתמשת בפונקציה והיא נועדה על מנת לעשות את ההשוואה בינה לבין השיטה Best Fit.

```python
def genome_rearrange(self, genome, bin_capacity):
    all_bins = [list(bin) for bin in genome]
    bin_loads = [(sum(b), idx) for idx, b in enumerate(all_bins)]
    bin_loads.sort(reverse=True)

    keep_count = len(bin_loads) // 2
    keep_indices = set(idx for _, idx in bin_loads[:keep_count])
    kept_bins = [all_bins[i] for i in range(len(all_bins))
                 if i in keep_indices]
    repack_items = [item for i in range(len(all_bins))
                    if i not in keep_indices for item in all_bins[i]]

    random.shuffle(repack_items)

    for item in repack_items:
        best_bin_index = -1
        min_space_left = bin_capacity + 1

        for i, bin in enumerate(kept_bins):
            space_left = bin_capacity - sum(bin)
            if space_left >= item and space_left - item < min_space_left:
                best_bin_index = i
                min_space_left = space_left - item

        if best_bin_index == -1:
            kept_bins.append([item])
        else:
            kept_bins[best_bin_index].append(item)

    return kept_bins
```

הפונקציה היא המקבילה ל- Crossover בבעיית ה- Bin Packing. היא מקבלת גנום, כלומר אוסף של פחים (Bins) ומבצעת עליו סידור מחדש באופן שהוסבר מקודם, כאשר היא שומרת את 50% הפחים עם התכולה הגבוהה ביותר כפי שהם, ומערבבת את כל האיברים שנמצאים בפחים האחרים ומארגנת אותם מחדש באמצעות Best Fit לתוך הפחים הקיימים או לפחים חדשים.

דוגמת הרצה:

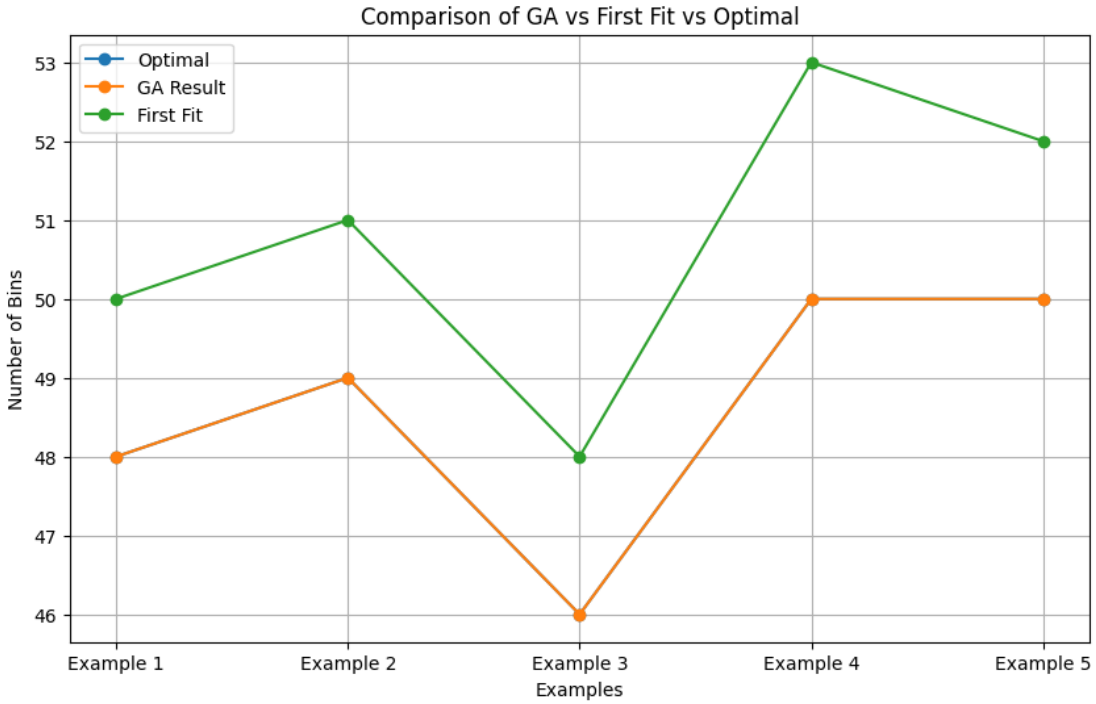
```
Running Example 1
Gen0. Best:  (49)  Worst: (54)   Fitness Range: 5   Avg: 50.99   Std: 0.71
    Ticks CPU: 0.0021, Elapsed: 0.0020s
Gen1. Best:  (49)  Worst: (54)   Fitness Range: 5   Avg: 50.38   Std: 0.66
    Ticks CPU: 0.6939, Elapsed: 0.7104s
Gen2. Best:  (49)  Worst: (53)   Fitness Range: 4   Avg: 50.15   Std: 0.68
    Ticks CPU: 1.3504, Elapsed: 1.3664s
Gen3. Best:  (49)  Worst: (53)   Fitness Range: 4   Avg: 50.03   Std: 0.68
    Ticks CPU: 2.0162, Elapsed: 2.0309s
Gen4. Best:  (49)  Worst: (54)   Fitness Range: 5   Avg: 50.00   Std: 0.72
    Ticks CPU: 3.1271, Elapsed: 3.1671s
Gen5. Best:  (49)  Worst: (54)   Fitness Range: 5   Avg: 49.96   Std: 0.71
    Ticks CPU: 4.2927, Elapsed: 4.3785s
Gen6. Best:  (49)  Worst: (53)   Fitness Range: 4   Avg: 49.91   Std: 0.69
    Ticks CPU: 5.0668, Elapsed: 5.1557s
Gen7. Best:  (49)  Worst: (53)   Fitness Range: 4   Avg: 49.91   Std: 0.71
    Ticks CPU: 5.7762, Elapsed: 5.9693s
Gen8. Best:  (49)  Worst: (53)   Fitness Range: 4   Avg: 49.89   Std: 0.72
    Ticks CPU: 6.4761, Elapsed: 6.7882s
Gen9. Best:  (49)  Worst: (53)   Fitness Range: 4   Avg: 49.89   Std: 0.71
    Ticks CPU: 7.1445, Elapsed: 7.4555s
Gen10. Best:  (49)  Worst: (53)   Fitness Range: 4   Avg: 49.87   Std: 0.72
    Ticks CPU: 7.8283, Elapsed: 8.1385s
Gen11. Best:  (49)  Worst: (53)   Fitness Range: 4   Avg: 49.87   Std: 0.71
    Ticks CPU: 8.5097, Elapsed: 8.8198s
Gen12. Best:  (49)  Worst: (53)   Fitness Range: 4   Avg: 49.87   Std: 0.73
    Ticks CPU: 9.1748, Elapsed: 9.4844s
Gen13. Best:  (48)  Worst: (53)   Fitness Range: 5   Avg: 49.87   Std: 0.72
    Ticks CPU: 9.8571, Elapsed: 10.1657s
Global optimum found!
```

אנחנו יכולים לראות בדור הראשון שה- bin הטוב ביותר היה בעל פיטניס 49, ואחרי כך הוא ניסה לכמה דורות למצוא ה- bin הטוב ביותר, ובדור ה 13 הוא התכנס לתשובה הנכונה.

האלגוריתם יכול להסתיים בשני מקרים:

1. סיום הזמן שמוקצה לו ובכך מתכנס לתשובה הכי טובה שהגיע עד כה.
2. התכנסות לתשובה האופטימלית.

תוצאות **של ההשוואה בין מספר הפחים בפתרון האופטימלי, באלגוריתם שלנו תוך שימוש ב- Best Fit וב- First Fit**:



בניתוח התוצאות שהתקבלו, ניתן לראות שהאלגוריתם הגנטי השיג תוצאות מרשימות במיוחד בכל אחת מחמש הדוגמאות שנבדקו, הוא הצליח להגיע בדיוק לפתרון האופטימלי שהוגדר מראש. כלומר, מספר הפחים שנמצא זהה לחלוטין למספר הפחים המינימלי האפשרי. עם זאת, חשוב לציין את ה Tradeoff-המרכזי First Fit: מהיר ופשוט אך לרוב פחות מדויק ודורש יותר פחים .האלגוריתם הגנטי איטי יותר, אך מספק פתרונות איכותיים בהרבה ובמקרה הזה, אפילו אופטימליים לחלוטין.

הערה: אחרי שהצלחנו להגיע למספר הפחים האופטימלי מקובץ binpack1.txt נסינו להריץ דוגמאות מתוך קבצים אחרים, האלגוריתם שלנו התכנס למספר קרוב מאוד למספר האופטימלי, הסיבה מכך כנראה היא שהדוגמאות בקבצים האחרים הן בעלות סדר גודל גבוהה יותר מהבעיות בקובץ שלנו. (גדול יותר מבחינת מספר האיברים וגם מספר הפחים האופטימלי)

<u>סעיף 12</u>

הוספה האפשרות לבחור את סוג הבעיה ע"י המשתנה הגלובלי PROBLEM.

לייצוג הפרטים והאוכלוסייה בבעיה השתמשנו באותם מחלקות שנבנו עבור בעיית "מחרוזת היעד". ערכי הפרטים שניתנים כמטריצה עוברים המרה למחרוזת המכילה את הערכים של

תאי המטריצה החל מהשמאלי לימני, שורה אחרי שורה. ייצוג זה נבחר על מנת לאפשר שימוש באותם פונקציות שכבר נבנו עבור בעיית המחרוזת כמו פונקציות חישוב הפיטניס, השיחלוף, הזיווג והמוטציות ועוד. דוגמה לכך:

המטריצה הבאה:

$$[[7,0,7],[7,0,7],[7,7,0]]$$

הופכת לייצוג: "707707770".

כמו כן, תכונות הפרטים בבעיה, בעלי צורה ריבועית, אפשרו את המעבר חזרה לצורה המטריציונית בלי הצורך לשמור את המבנה שלה.

למעבר מהצורה המטריציונית למחרוזת הוספנו את הפונקציה הבאה:

```
def matrix_to_string(self, matrix):
    string = []
    for row in matrix:
        string.extend(row)
    return "".join(str(num) for num in string)
```

בשל הצורך לעשות את המעבר מהייצוג כמחרוזת בחזרה למטריצה דו-ממידת, למשל, במקרה של הדפסה נרצה שזה יודפס כמטריצה ולא כמחרוזת. הדבר נעשה ע"י הוספת הפונקציה הבאה:

```
def string_to_matrix(self, string):
    size = int(math.sqrt(len(string)))
    matrix = []
    for i in range(size):
        row = [int(c) for c in string[i *
                    size:(i + 1) * size]]
        matrix.append(row)
    return matrix
```

שתי הפונקציות, המאתחלת את האוכלוסייה באופן רנדומלי והיוצרת מוטציה, הותאמו בכך שבמקרה הבעיה הזאת הם יגרילו אותיות רק מהטווח 0-9 (ערכי ה- ASCII בתחום 48-57). למרות שהשינוי הזה לא היה נחוץ הוא נעשה מטעמי יעילות.

בנוסף, נוספה האפשרות לקבל את הגינום ההתחלתי כארגומנט. במקרה הזה, האוכלוסייה ההתחלתית תהיה תאותחל כולה כבעלת אותו גינום התחלתי.