מעבדה בבינה מלאכותית

דו"ח תרגיל בית 2 – Evolution Control

שמות:

עוביידה חטיב, 201278066

אסיל נחאס, 212245096

סביבת ההרצה והכלים בהם נעשה שימוש

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

הרצת קובץ הפייתון

קוד הפייתון רץ באמצעות הפקודה הבאה:

```
python lab2.py <time_limit> <problem_type> <file_path>
```

כאשר:

- time_limit: זמן ההרצה המקסימלי. במקרה שזמן ההרצה מגיע לסף הזה ההרצה מפסיקה והגינום של הפרט בעל הפיטניס הטוב ביותר שהתקבל עד אז יודפס.
- problem_type: סוג הבעיה מבין שתי הבעיות בהם הקוד שלנו מטפל - TSP לבעיית Travelling Salesman, ו- BIN_PACK לבעיית ה- Bin Packing.
- file_path: הנתיב לקובץ הקלט של הבעייה.

דוגמאות לפקודות הרצה חוקיות:

```
python lab2.py 300 TSP st70.tsp

python lab2.py 500 BIN_PACK binpack1.txt
```

הרצת קובץ ה- EXE

קוד ה- EXE רץ באמצעות הפקודה הבאה:

```
lab2.exe <time_limit> <problem_type> <file_path>
```

כאשר הארגומנטים שהוא מקבל הם אותם ארגומנטים שהוסבר לגבם בהסבר לגבי הרצת קובץ הפייתון.

ייצוג הפרטים והאוכלוסייה

- **בעיית ה- DTSP**: הגינום של הפרט מיוצג ע"י שתי תמורות המייצגים שני מסלולים הזרים בקשתות. המסלול הראשון מאותחל באופן אקראי, כך גם השני רק שהוא נבדק מול הראשון, ואם משתתף אתו בקשת הוא מאותחל מחדש עד שמתקבל אחד שהוא זר.
  חישוב אורך המסלול מתבצע ע"י סכימת המרחקים בין קואורדינטות של כל עיר והעיר שאחריה, החל מהעיר הראשונה ועד החזרה לאותה עיר. הפיטניס של פרט מוגדר כאורך המסלול הארוך מבין שני המסלולים בגנום שלו.
- **בעיית ה- Bin Packing**: כל משקל קיבל אינדקס ייחודי ואחיד בתוך האוכלוסייה, והפרט יוצג ע"י תמורה של אינדקסי המשקלים. הבחירה להתמודד עם אינדקסים במקום להתעסק באופן ישיר עם המשקלים נבחרה בגלל שמשקלים יכולים להיות דומים בערכם מה שלא יאפשר ייצוג הפרט כתמורה ובכך לא נוכל להשתמש בשיטות השיחלוף השונות שיודעות להתמודד עם תמורות.
  חישוב הפיטניס של הפרט התבצע ע"י מספר הפחים (Bins) שפרט שצריך פחות מספר הפחים בפתרון האופטימלי. חישוב הפחים שהפרט צריך חושב ע"י לעבור על המשקלים של התמורה בסדר ולהכניס אותם לפחים לפי שיטת ה- Best Fit.

בחירת הפרמטרים

הערכים ההתחלתיים של גודל האוכלוסייה, אחוז האליטה (Elitism rate), ההסתברות למוטציה (Mutation Rate), מספר האיטרציות של ההתכנסות המקומית, ושיטת בחירת ההורים נבחרו כאלה שהביאו לתוצאות הכי טובות במעבדה הקודמת. חלק מהפרמטרים השתנו בהמשך ופרמטרים אחרים התווספו כתלות במשימה כפי שיוסבר בסעיפים הרלוונטיים.

<u>סעיף 1</u>

הקוד שלנו מתחיל בבדיקת תקינות הקלט, כפי שהוסבר למעלה. המשתנים הגלובליים של הזמן המקסימלי לריצה וסוג הבעיה מאותחלים לכאלה שהתקבלו כקלט. כמו כן, הנתונים

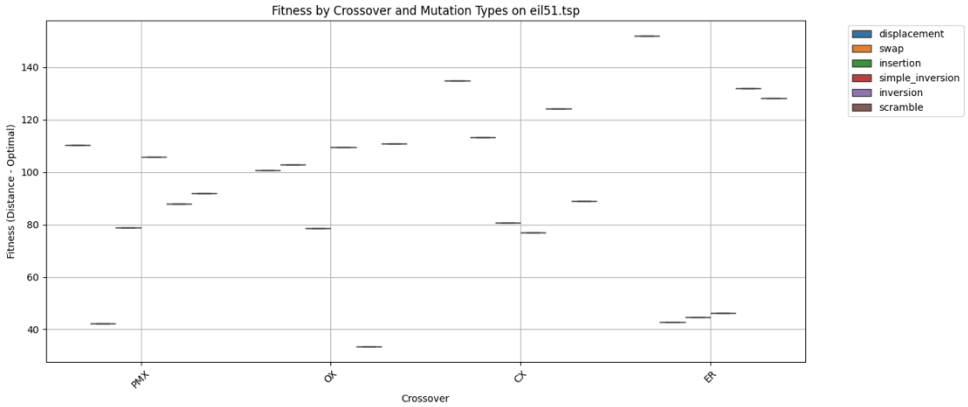
הרלוונטיים לבעיה מחולצים מתוך הקובץ של הקלט, כאשר במקרה של בעיית ה- DTSP הנתונים כוללים את הקואורדינטות של הערים ואת הפתרון האופטימלי, ובמקרה של בעיית ה- Bin Packing הם כוללים את המשקלים השונים, גודל הפח, והפתרון האופטימלי. לאחר מכן, מופע האוכלוסייה והמופעים של הפרטים בה מאותחלים באופן רנדומלי כתמורות בגודל הערים\המשקלים. האלגוריתם רץ עד שאחד מ- 3 הבאים מתקיים:

1. מתכנס באופן גלובלי: תנאי זה מאותר ע"י כך שהפיטניס הטוב ביותר הוא 0.
2. מתכנס באופן לוקאלי: תנאי זה מאותר ע"י כך שהפיטניס הטוב ביותר לא משתפר במשך 50 איטרציות.
3. זמן ההרצה אזל: תנאי זה מאותר ע"י כך שהזמן שעבר מאז תחילת הרצת האלגוריתם שווה או גדול מהזמן המקסימלי שהתקבל כקלט.

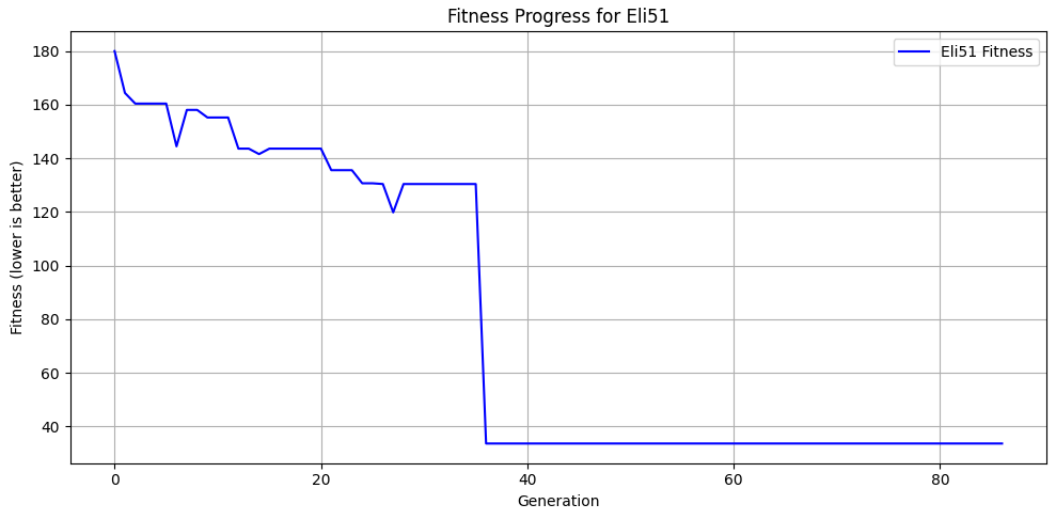
בכל אחד מהאיטרציות\הדורות מתבצעות הפעולות הבאות: הפיטניס של הפרטים מחושב מחדש, הסטטיסטיקה של האוכלוסייה (הפיטניס הטוב ביותר, ממוצע הפיטניסים וסטיית התקן שלהם) מחושבת, ונבחרת האוכלוסייה של הדור הבא כצירוף של האליטה (5% הפרטים בעלי הפיטניס הטוב ביותר באוכלוסייה), ופרטים החדשים הנוצרים כתוצאה של שיחלוף בין שני הורים מהדור הנוכחי. כמו כן, הפרטים עוברים מוטציה בסבירות של 25%.

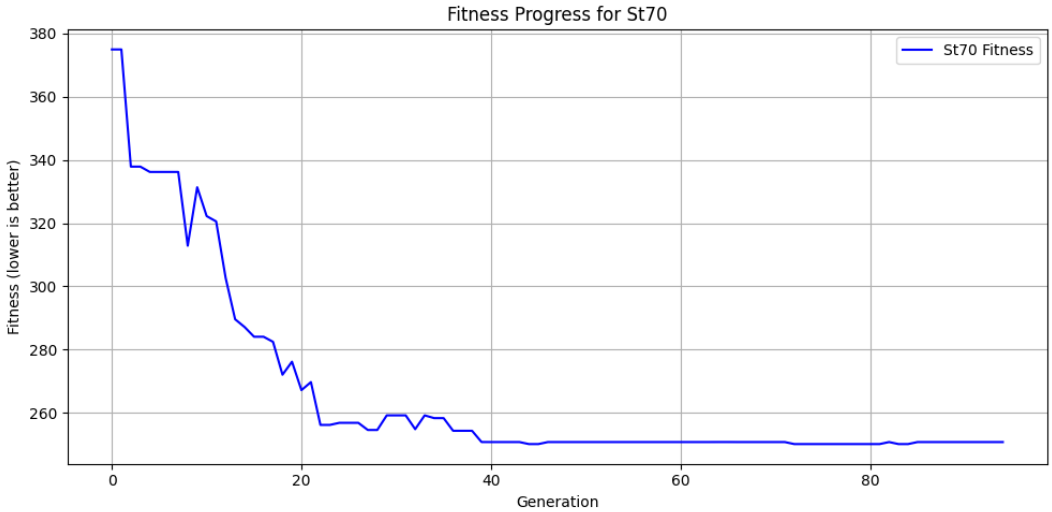
שיטת השיחלוף והמוטציות נבחרו מתוך 24 של זוגות (4 שיטות שחלוף ו- 6 שיטות המוטציות שלמדנו). לכל זוג התבצעו 4 הרצות, והפיטניס הטוב ביותר התמצע על פניהם. הזוג הטוב ביותר נבחר כזה שהביא לממוצע הנמוך ביותר, והיה (OX, Inversion) ועל כן הוא אומץ.

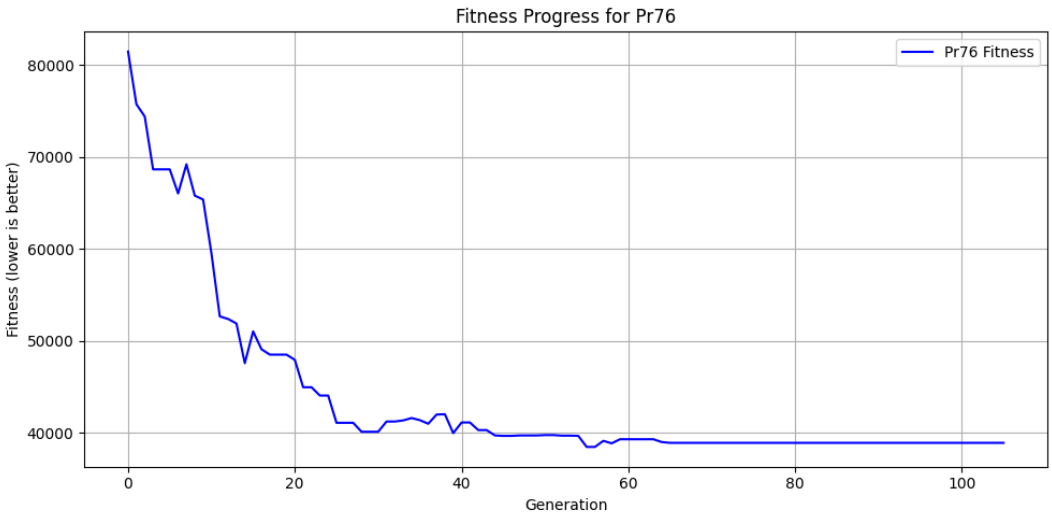
	Crossover	Mutation	AvgFitness
14	OX	inversion	33.58
23	PMX	swap	42.40
11	ER	swap	42.67
7	ER	insertion	44.58
10	ER	simple_inversion	46.24
4	CX	simple_inversion	77.08
13	OX	insertion	78.47
19	PMX	insertion	78.82
1	CX	insertion	80.72
20	PMX	inversion	87.90
3	CX	scramble	89.00
21	PMX	scramble	92.01
12	OX	displacement	100.62
17	OX	swap	102.96
22	PMX	simple_inversion	105.80
16	OX	simple_inversion	109.54
18	PMX	displacement	110.37
15	OX	scramble	110.96
5	CX	swap	113.23
2	CX	inversion	124.09
9	ER	scramble	128.16
8	ER	inversion	131.89
0	CX	displacement	134.81
6	ER	displacement	151.85

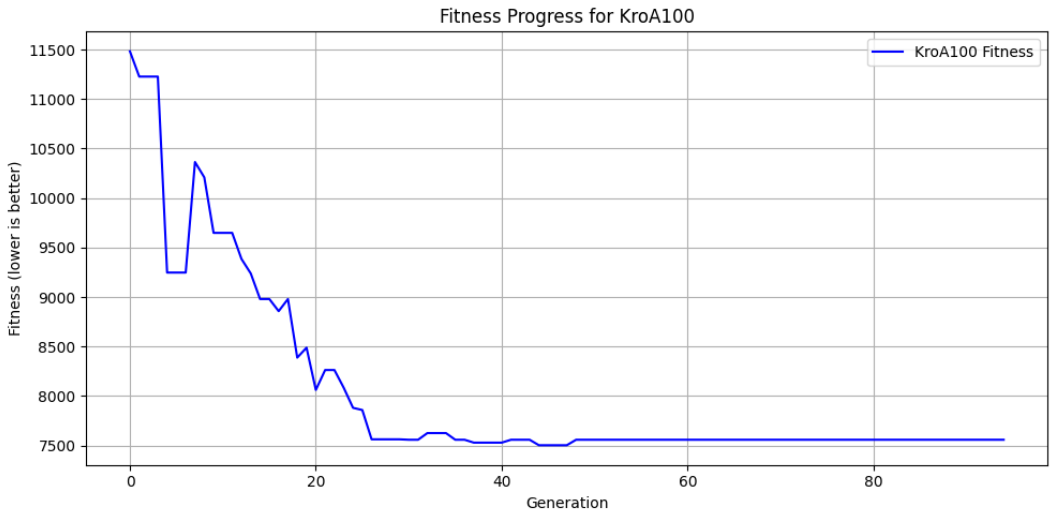


להלן תוצאות הרצת האלגוריתם על כל אחת מ- 4 הבעיות המצוינות בסעיף. התוצאות מציגות את הפיטניס של הפרט הטוב ביותר לפי הדור:


Fitness Progress for Eli51
Eli51 Fitness
Fitness (lower is better)
Generation
180
160
140
120
100
80
60
40
0
20
40
60
80


Fitness Progress for St70
St70 Fitness
Fitness (lower is better)
Generation
380
360
340
320
300
280
260
0
20
40
60
80


Fitness Progress for Pr76
Pr76 Fitness
Fitness (lower is better)
Generation
80000
70000
60000
50000
40000
0
20
40
60
80
100



סעיף 2

א. מימוש שתי המדיניות בוצעו ע"י שתי הפונקציות המצורפות למטה כחלק ממחלקת האוכלוסייה הראשית BasePopulation. כאשר בפונקציה של המדיניות הלא לינראית קצבי המוטציות המינימלי והמקסימלי היו 10% ו- 50% בהתאמה, והערך של הפרמטר r היה 0.05% ואילו במדיניות ה- triggered hypermutation אחוז הקצב הנמוך עמד על 25% (כמו זה של ברירת המחדל) והקצב הגבוה היה 50%. ב- triggered hypermutation נוסף עוד פרמטר של המשתנה על פיו ההחלטה לגבי השימוש בקצב הגבוה או הנמוך מתבצעת מבין 3 האופציות: הפרט בעל הפיטניס הטוב ביותר (BEST_FIT), הממוצע של הפיטניסים (AVG_FIT), וסטיית התקן של הפיטניסים (STD_FIT). הערך של המשתנה הנבחר נבדק מול הערך הטוב ביותר שהתקבל ב- k האיטרציות האחרונות ובמקרה שהוא לא טוב ממנו, הקצב המאומץ הופך לזה הגבוה. הקצב נשאר גבוה עד לשיפור של 1%.
הערכים של הפרמטרים נעשו לפי ניסויים שנעשו תוך שימוש בקומבינציות שונות של ערכים, והקומבינציה בעלת המספרים האידאליים מבחינת הפיטניס הנמוך ביותר היא שנבחרה עבור כל שיטה.

```python
def non_linear_mutation_policy(self):
    global GA_MUTATIONRATE

    min_mutation_rate=0.05
    p_max=0.5
    r=0.01
    generation = len(self.average_fitness)

    exp_term = math.exp(-r * generation)
    numerator = 2 * (p_max * exp_term)
    denominator = 1 + exp_term
    f_t = numerator / denominator

    GA_MUTATIONRATE = max(f_t , min_mutation_rate)
```

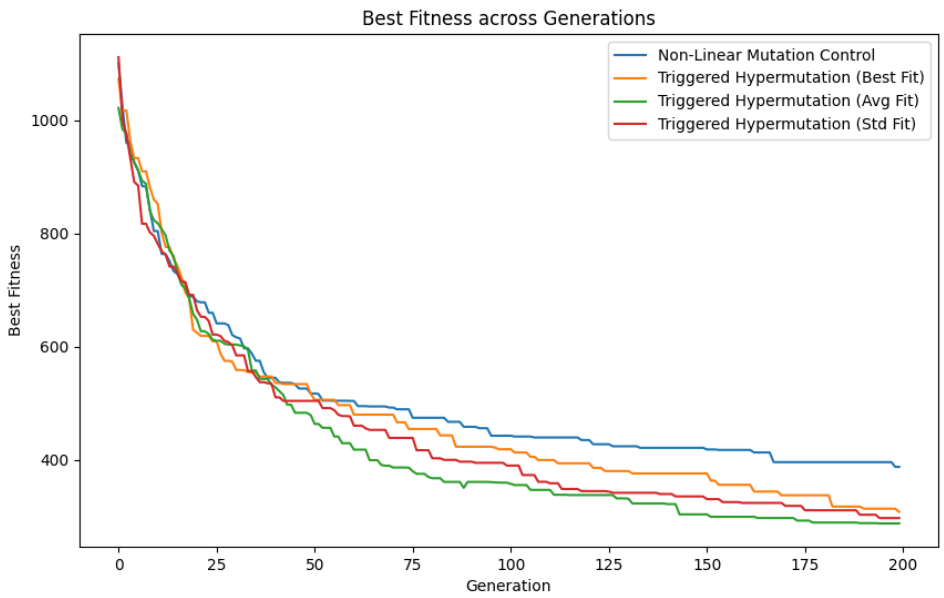
```python
def trigger_hyper_mutation_policy(self):
    global GA_MUTATIONRATE, TRIG_HYPER_TRIGGER, HIGH_MUTATION_START_VAL
    high_mutation = 0.5
    low_mutation = 0.25
    k = 10
    if TRIG_HYPER_TRIGGER == "BEST_FIT":
        if HIGH_MUTATION_START_VAL and HIGH_MUTATION_START_VAL -
            self.best_fitness[-1] >= HIGH_MUTATION_START_VAL * 0.01:
            GA_MUTATIONRATE = low_mutation
            HIGH_MUTATION_START_VAL = None
        elif HIGH_MUTATION_START_VAL:
            return
        elif len(self.best_fitness) > k and self.best_fitness[-1]
                                    >= min(self.best_fitness[-k:]):
            GA_MUTATIONRATE = high_mutation
            HIGH_MUTATION_START_VAL = self.best_fitness[-1]
    elif TRIG_HYPER_TRIGGER == "AVG_FIT":
        if HIGH_MUTATION_START_VAL and HIGH_MUTATION_START_VAL -
            self.average_fitness[-1] >= HIGH_MUTATION_START_VAL * 0.01:
            GA_MUTATIONRATE = low_mutation
            HIGH_MUTATION_START_VAL = None
        elif HIGH_MUTATION_START_VAL:
            return
        elif len(self.average_fitness) > k and self.average_fitness[-1]
            - self.average_fitness[-k] < min(self.average_fitness[-k:]):
            GA_MUTATIONRATE = high_mutation
    elif TRIG_HYPER_TRIGGER == "STD_FIT":
        if HIGH_MUTATION_START_VAL and HIGH_MUTATION_START_VAL
            - self.std_devs[-1] >= HIGH_MUTATION_START_VAL * 0.01:
            GA_MUTATIONRATE = low_mutation
            HIGH_MUTATION_START_VAL = None
        elif HIGH_MUTATION_START_VAL:
            return
        elif len(self.std_devs) > k and self.std_devs[-1]
                - self.std_devs[-k] < min(self.std_devs[-k:]):
            GA_MUTATIONRATE = high_mutation
```

בהתאם לנדרש בסעיף, נעשתה השוואה בין 4 המדיניות הבאות של קצב המוטציות:

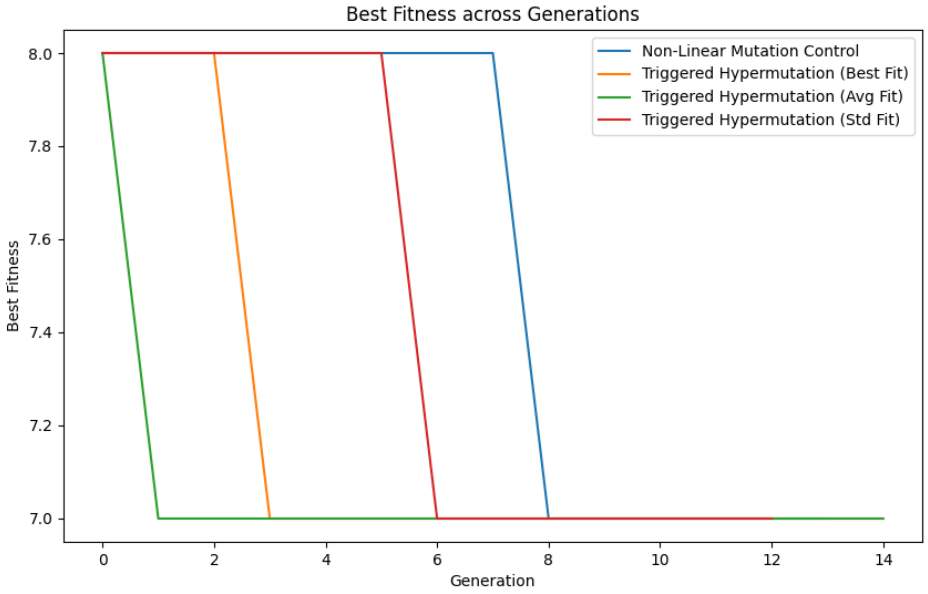
1. מדיניות לא לינארית.
2. מדיניות Triggered Hypermutation כאשר הטריגר הוא אי-השיפור בפיטניס הטוב ביותר.
3. מדיניות Triggered Hypermutation כאשר הטריגר הוא אי-השיפור בממוצע הפיטניסים של האוכלוסייה.
4. מדיניות Triggered Hypermutation כאשר הטריגר הוא אי-השיפור בסטיית התקן של הפיטניסים של האוכלוסייה.

הפרמטרים האחרים נשארו זהים, תוך שימוש בקונפיגורציה הטובה ביותר שהתקבלה עד כה, בנוסף לשימוש בפונקציה random.seed המבטיחה התנהגות אקראית שווה.

התוצאות שהתקבלו עבור בעיית ה- DTSP:



התוצאות שהתקבלו עבור בעיית ה- Bin Packing:



מדיניות ה- Triggered Hypermutation הראתה אפקטיביות יותר על המדיניות הלא-לינארית, בפרט לפי טריגר השינוי בממוצע הפיטניסים. האפקט נראה יותר ברור בבעיית ה- DTSP שם הוא התבטא בערכים יותר נמוכים לפיטניס הטוב ביותר. בבעיית ה- Bin Pack זה התבטא במהירות ההתכנסות (לפי דור), אולם בסיום האלגוריתם הם הגיעו לאותה תוצאה.

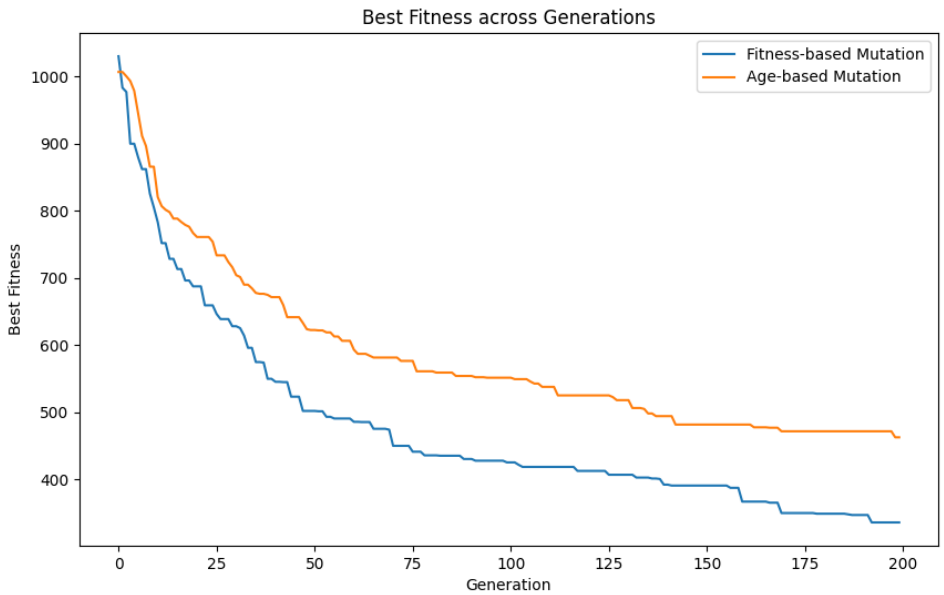
ב. מימוש שתי המדיניות בוצע ע"י שתי הפונקציות הבאות כחלק ממחלקת האוכלוסייה הראשית BasePopulation. בשניהם קצב המוטציה המינימלי נהיה 5%.

```
def fit_based_ind_mutation(self):
    avg_fit = sum(ind.fitness for ind in self.individuals) /
                                        len(self.individuals)
    for ind in self.individuals:
        rel_fit = ind.fitness / avg_fit if avg_fit > 0 else 1
        mut_rate = max(0.05, GA_MUTATIONRATE * (1 - rel_fit))
        if random.random() < mut_rate:
            ind.mutate()
```

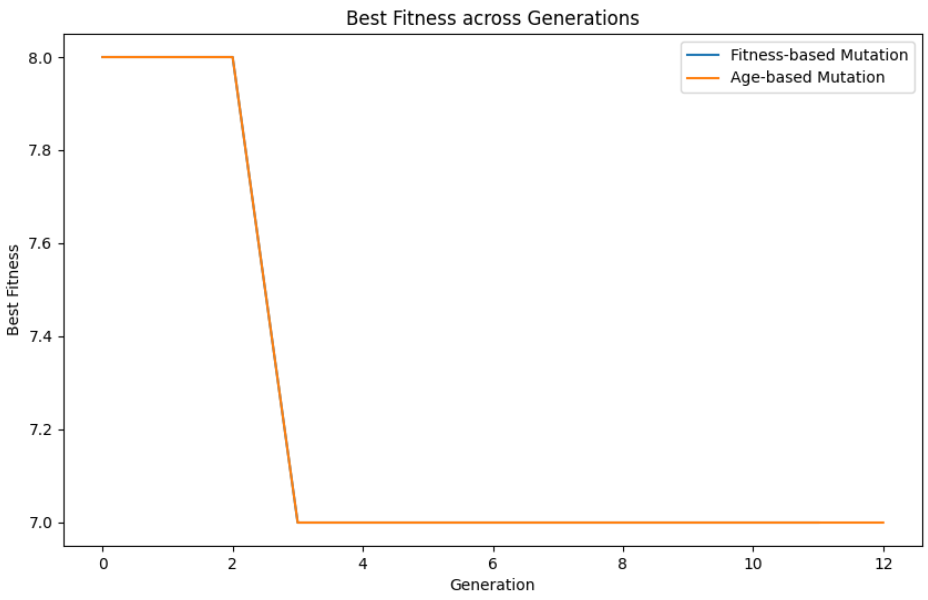
```python
def age_based_ind_mutation(self):
    p_min = 0.05
    alpha = 0.05
    for ind in self.individuals:
        mutation_rate = min(GA_MUTATIONRATE, p_min + alpha
                                                    * ind.age)
        if random.random() < mutation_rate:
            ind.mutate()
```

כמו במקרה המדיניות בסעיף א' גם כאן ערכי הפרמטרים נבחרו לאחר תהליך של Tuning, ובאותה מידה ההשוואות נעשו תוך שימוש באותם פרמטרים לא קשורים למטלה, ובפונקציית random.seed.

התוצאות שהתקבלו עבור בעיית ה- DTSP:



התוצאות שהתקבלו עבור בעיית ה- Bin Pack:



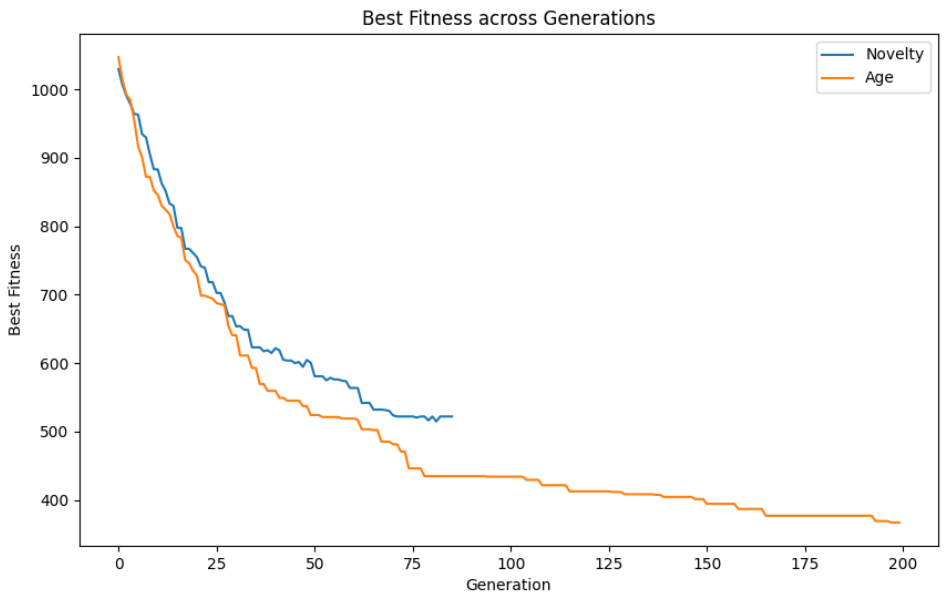
לפי התוצאות של ה- DTSP, מוטציות מבוססות פיטניס משיגים שיפור מהיר וגדול יותר לעומת המוטציות מבוססות גיל, מה שמצביע על כך שהיא מצליחה באופן יותר יעיל בלפתור את בעיית ההתכנסות לאופטימום מקומי ובהנעה לכיוון פתרונות חדשים. בהשוואה לפי בבעיית ה- Bin Pack לא התקבל הבדל בין שתי השיטות, גם בשימוש בקבצים שונים. גם ההשוואה שכללה אי-שימוש בשיטה כלשהי לא הניבה תוצאות שונים באופן עקבי, מה שמצביע על כך שהיא לא בעלת השפעה משמעותית בבעיה הספציפית הזאת.

ג. מימוש חישובי הפיטניס האינדיבידואלי מבוסס החדשנות ומבוסס הגיל בוצע ע"י שתי הפונקציות המצורפות למטה, שהתווספו לפונקציה evaluate_fitness המחשבת את הפיטניס של הפרטים, ומתבצעות לאחר חישוב הפיטניס הסטטי:

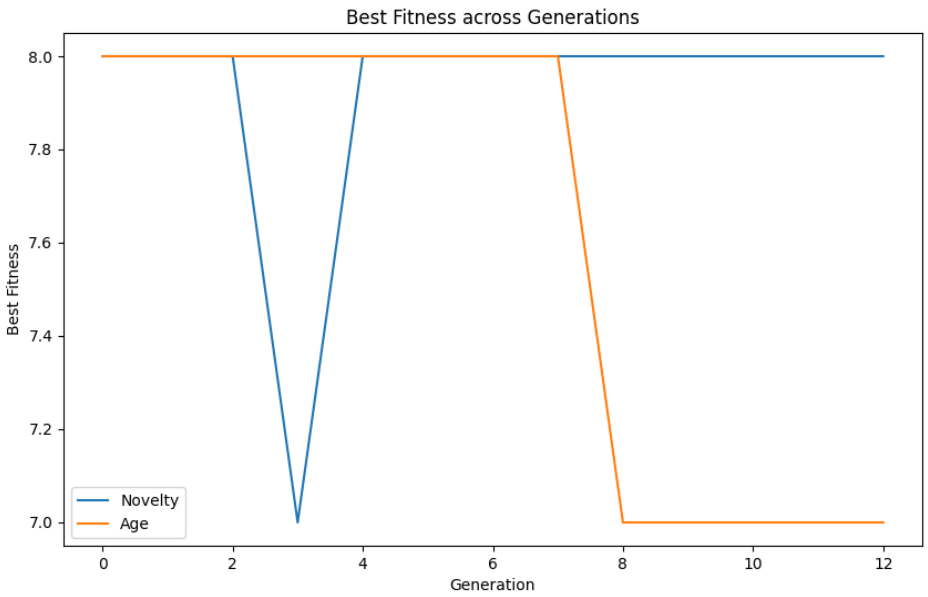
```python
if FITNESS_COMPUTATION_MODE == "NOVELITY":
    edges = []
    for ind in self.individuals:
        route_edges = set()
        for j in range(len(ind.genome[0])):
            a1 = ind.genome[0][j]
            b1 = ind.genome[1][(j + 1) % len(ind.genome)]
            route_edges.add((a1, b1))
            route_edges.add((b1, a1))
            a2 = ind.genome[1][j]
            b2 = ind.genome[0][(j + 1) % len(ind.genome)]
            route_edges.add((a2, b2))
            route_edges.add((b2, a2))
        edges.append(route_edges)
    for j, ind in enumerate(self.individuals):
        neighbors = []
        for i in range(len(edges)):
            intersection = len(edges[i].intersection(edges[j]))
            neighbors.append((i, intersection))
        neighbors.sort(key=lambda x: x[1], reverse=True)
        k_neighbors = neighbors[1:NOVELITY_K+1]
        novelty_score = - sum([x[1] for x in k_neighbors])
                                                / NOVELITY_K
        ind.fitness = WEIGHT_FACTOR * ind.fitness +
                        (1 - WEIGHT_FACTOR) * novelty_score

elif FITNESS_COMPUTATION_MODE == "AGE":
    for ind in self.individuals:
        ind.fitness = WEIGHT_FACTOR * ind.fitness +
                        (1 - WEIGHT_FACTOR) * ind.age
```

ההשוואה נעשתה בדומה לשני הסעיפים א' ו- ב' על סמך הפיטניס הטוב ביותר באוכלוסייה תוך שימוש בפיטניס הסטטי לשם ההשוואה. מלבד שיטת הפיטניס האינדוודאלי שאר הפרמטרים היו משותפים לשתי הריצות.

התוצאות עבור בעיית ה- DTSP:



התוצאות עבור בעיית ה- Bin Pack:



במקרה של בעיית ה- DTSP, למרות שחישוב הפיטניס האינדיבידואלי מבוסס חדשנות מראה שיפור ראשוני דומה ואפילו טוב יותר מהמבוסס גיל, האחרונה פותחת פער גדול יותר ככל שהדורות מתקדמים. כמו כן, ובאופן צפוי, השיטה המבוססת פיטניס מצריכה יותר חישוב ומוסיפה סיבוכיות גדולה לעומת המבוססת גיל, דבר המשתקף בהפרש במספר

הדורות שהתבצעו בין שתי השיטות במהלך אותו פרק זמן. התוצאות של בעיית ה- Bin Packing לא מראות הבדל משמעותי.

סעיף 3

עבור כל אחת משתי הבעיות מומשה פונקציית דמיון, הפונקציה מקבלת כפרמטר שני פרטים ומחזירה ערך שמייצג את הדמיון בין שני הגינומים שלהם.

עבור בעיית ה- DTSP, הפונקציה מחשבת את הדמיון ע"י כך שהיא מחלצת את הקשתות של המסלול הארוך (הראשון מבין השניים, מכיוון שהם מסודרים בזמן חישוב הפיטניס כך שהראשון הוא הארוך) של כל אחד משני הפרטים, ומוצאת החיתוך של שניהם. הפונקציה לאחר מכן מנרמלת את המספר ע"י החלוקה במספר הקשתות הכללי במסלול ומחזירה אחוז שבין 0 ל- 1.

```python
def distance_between_genomes_TSP(ind1, ind2):
    def get_edges(tour):
        return set((min(tour[i], tour[(i + 1) % len(tour)]),
                    max(tour[i], tour[(i + 1) % len(tour)]))
                                    for i in range(len(tour)))

    edges1 = get_edges(ind1.genome[0]).union(get_edges(ind1.genome[1]))
    edges2 = get_edges(ind2.genome[0]).union(get_edges(ind2.genome[1]))

    shared_edges = len(edges1.intersection(edges2))
    total_edges = len(edges1.union(edges2))
    return 1 - shared_edges / total_edges
```

הפונקציה של בעיית ה- Bin Pack מחשבת את הדמיון ע"י השימוש באלגוריתם ההונגרי. קודם כל הפונקציה מאזנת את מספר הפחים של שני הפרטים ע"י כך שהיא מוסיפה פחים ריקים לפרט בעל מספר הפחים הקטן מבין השניים עד שהם שווים. לאחר מכן היא מטריצת עלויות בגודל של מספר הפחים כאשר כל תא במטריצה מייצג את מספר הפריטים השונים בין שני הפחים הרלוונטיים. ולבסוף, מחושב סכום ההבדלים בין הפחים בהתאמה שנבחרה והוא מנורמל לפי מספר הפריטים הכולל בשני הפחים כך שיהיה בין 0 ל- 1. ערך הדמיון המוחזר הוא 1 – "ההבדל המנורמל", כך ש- 1 הוא זהות מוחלטת ו- 0 הוא שוני מוחלט.

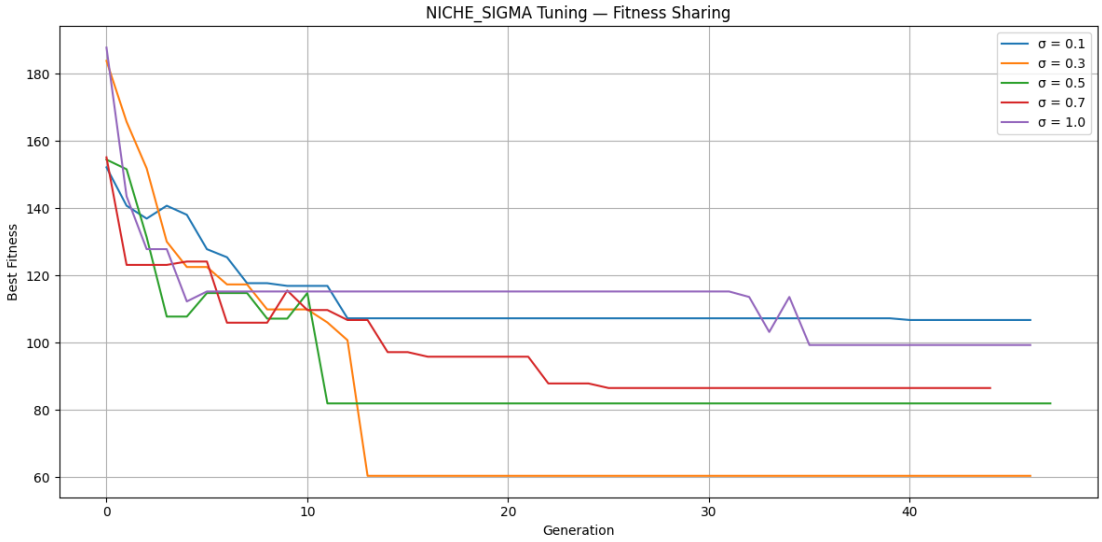
```python
def distance_between_genomes_BinPack(ind1, ind2, bin_capacity):
    bins1 = split_bins(ind1.genome, bin_capacity)
    bins2 = split_bins(ind2.genome, bin_capacity)

    size = max(len(bins1), len(bins2))
    while len(bins1) < size:
        bins1.append([])
    while len(bins2) < size:
        bins2.append([])

    cost_matrix = np.zeros((size, size))
    for i in range(size):
        for j in range(size):
            set1 = set(bins1[i])
            set2 = set(bins2[j])
            cost_matrix[i][j] = len(set1.symmetric_difference(set2))

    row_ind, col_ind = linear_sum_assignment(cost_matrix)
    total_diff = cost_matrix[row_ind, col_ind].sum()
    total_items = sum(len(b) for b in bins1)

    return total_diff / total_items if total_items > 0 else 0
```

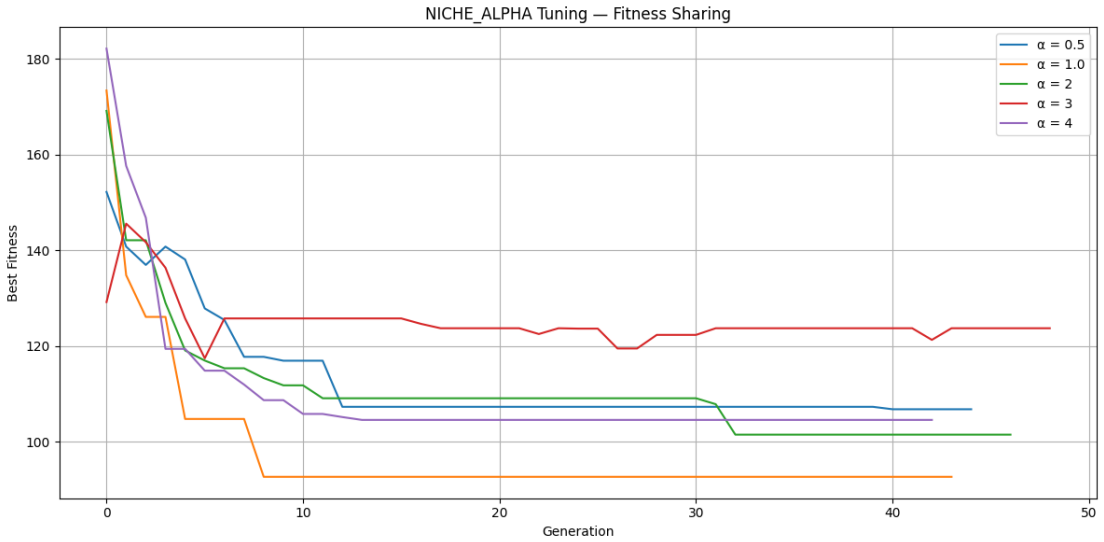
סעיף 4

את **אלגוריתם המחיצות** יושם ע"י הפונקציה המצורפת למטה, אשר קובעת את הפיטניס של הפרט על פי כמות הפרטים הדומים לו במטרה לעודד פרטים ייחודיים. היא עובדת ע"י כך שהיא מחשבת לכל פרט עד כמה הוא דומה לשאר הפרטים באמצעות פונקציית הדמיון שישמנו בסעיף הקודם. אם המרחקים בין שניהם קטן מסיגמה, היא משייכת אותם לאותה נישה, ותרומתו מצטברת ל- Sharing sum שלו, שככל שהוא גדול יותר זה מעיד על פרט שכיח יותר וכתוצאה מכך ניתנת לו פיטניס נמוך יותר.

```python
def apply_fitness_sharing(self):
    if not NICHE_METHOD_USE:
        return

    sigma = sigma if sigma is not None else NICHE_SIGMA
    alpha = alpha if alpha is not None else NICHE_ALPHA

    TOP_K = 40
    if len(self.individuals) <= TOP_K:
        subset = self.individuals
    else:
        subset = sorted(self.individuals, key=lambda ind: ind.fitness)[:TOP_K]

    distance_matrix = np.zeros((len(subset), len(subset)))

    for i in range(len(subset)):
        for j in range(i + 1, len(subset)):
            if PROBLEM == "TSP":
                dist = distance_between_genomes_TSP(subset[i], subset[j])
            elif PROBLEM == "BIN_PACK":
                dist = distance_between_genomes_BinPack(subset[i],
                    subset[j], self.bin_max)
            distance_matrix[i][j] = dist
            distance_matrix[j][i] = dist

    for i in range(len(subset)):
        sharing_sum = 0
        for j in range(len(subset)):
            dist = distance_matrix[i][j]
            if dist < sigma:
                sharing_sum += 1 - (dist / sigma) ** alpha
        sharing_sum = max(sharing_sum, 1e-6)
        subset[i].fitness = subset[i].raw_fitness * (1 / sharing_sum)
```

את הערכים של הסיגמה והאלפא נבחרו לאחר תהליך Tuning שסקר את ביצוע האלגוריתם תחת ערכים שונים של הפרמטרים, ולקח בחשבון גם את התוצאה הטובה ביותר שהתקבלה וגם את מספר הדורות שהאלגוריתם הצליח להריץ בשימוש באותו ערך. על סמך התוצאות המצורפות להלן נקבעו הערכים של סיגמה ואלפא להיות 0.3 ו- 1.0 בהתאמה.

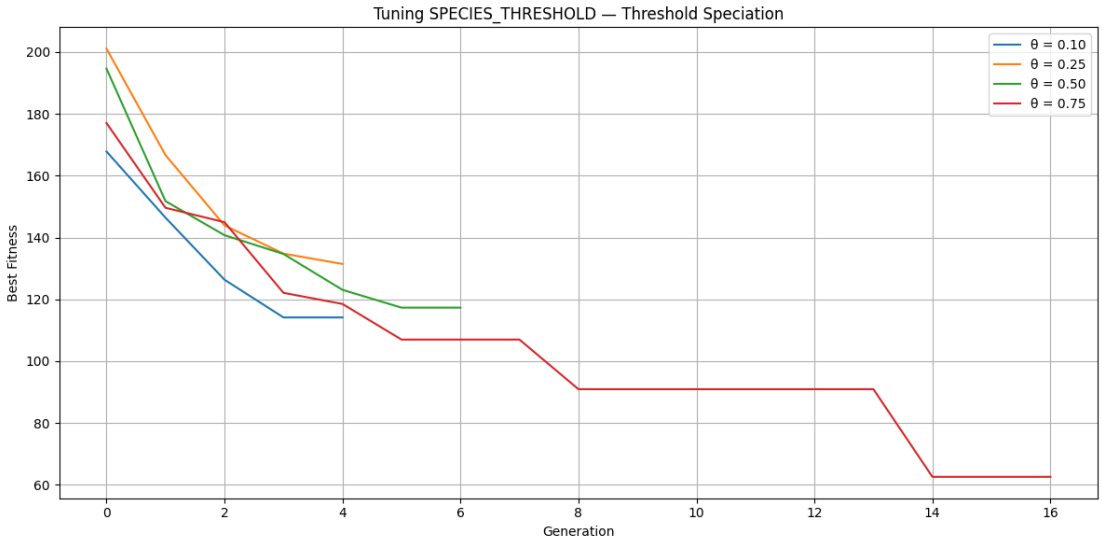




אלגוריתם פיצול הזנים מומש ע"י הפונקציה המצורפת למטה. היא מחלקת את האוכלוסייה לקבוצות לפי דמיון, שגם כן מחושב ע"י הפונקציה שמומשה בסעיף הקודם. הפונקציה מבצעת את החלוקה בצורה הבאה: מתחילה עם רשימה ריקה של זנים (כל אחת מיוצגת ע"י "הפרט המייסד"). עבור כל פרט הפונקציה בודקת אם הוא דומה לנציג של אחד הזנים הקיימים, אם כן היא מצרפת אותו לזן של הנציג הדומה, ואם לא נמצא נציג כזה, זן חדש נפתח והפרט נהיה נציג שלו. החלוקה נשמרת באמצעות מבנה נתונים של מלון, המאפשר לאחר מכן לאלגוריתם לעשות את השיחלוף בין שני הורים ששייכים לאותו זן. היא לאחר מכן מתאימה באופן דינמי את סף הדמיון (Species Threshold) על סמך מספר המינים שנוצרו באיטרציה, מעלה אותו כאשר מספר המינים הוא גדול מהמצופה ומקטינה אותו במקרה ההפוך.

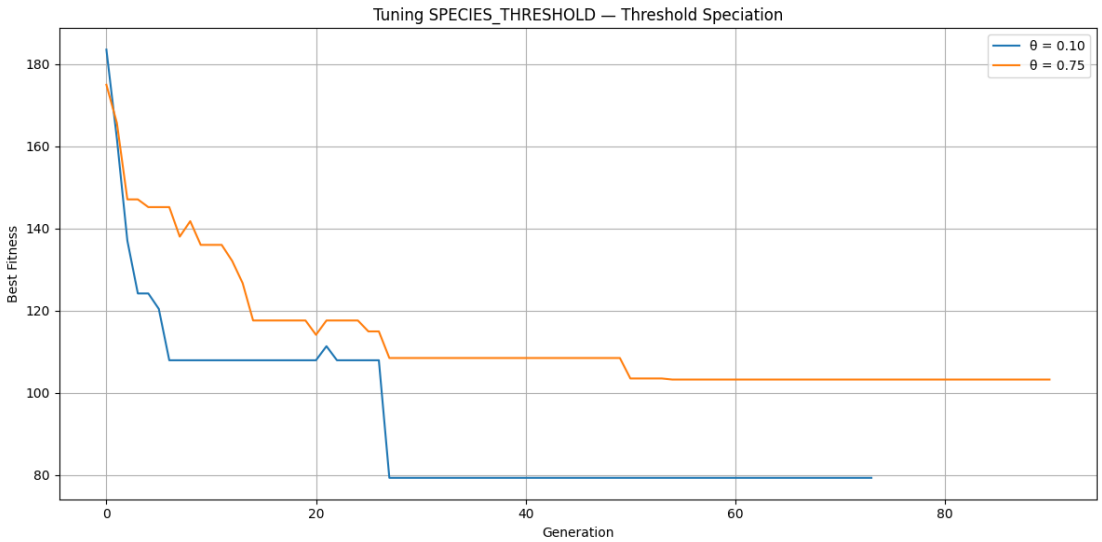
```python
def apply_threshold_speciation(self):
    global SPECIES_THRESHOLD

    self.species = {}
    for ind in self.individuals:
        placed = False
        for s in self.species:
            if distance_between_genomes_TSP(ind,
                self.species[s][0]) < SPECIES_THRESHOLD:
                self.species[s].append(ind)
                placed = True
                break
        if not placed:
            self.species[len(self.species)] = [ind]

        ind.specie = len(self.species) - 1

    current_count = sum(len(s) for s in self.species.values())

    if current_count < SPECIES_TARGET_COUNT:
        SPECIES_THRESHOLD += THRESHOLD_ADJUST_RATE * SPECIES_THRESHOLD
    elif current_count > SPECIES_TARGET_COUNT:
        SPECIES_THRESHOLD -= THRESHOLD_ADJUST_RATE * SPECIES_THRESHOLD
    SPECIES_THRESHOLD = max(SPECIES_THRESHOLD, 0.01)
```

את הערכים של הפרמטרים השונים של האלגוריתם, הכוללים את מספר המינים הרצוי, סף הדמיון (לקביעת שייכות לאותו מין), ומידת השינוי של הסף (Threshold Adjust Rate) גם הוא התבצע ע"י תהליך של Tuning. תהליך קביעת הערך המתאים לסף הדמיון היה מאתגר, ולהלן התוצאות שהתקבלו מהצרת האלגוריתם תוך שימוש בערכים שונים של הפרמטר:

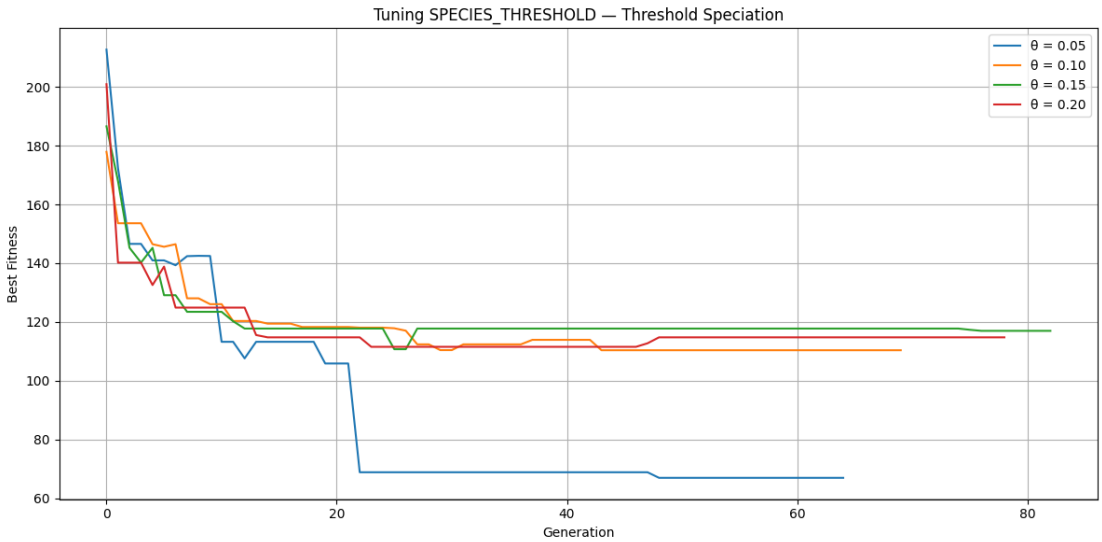


מהתוצאות לא היה ניתן לקבוע באופן וודאי לגבי הטוב מבין הערכים, אולם היו מספיקות על מנת לצמצם את מספר האפשרויות לשניים, 0.1 ו- 0.75, הראשון אומנם היה טוב מהשני,

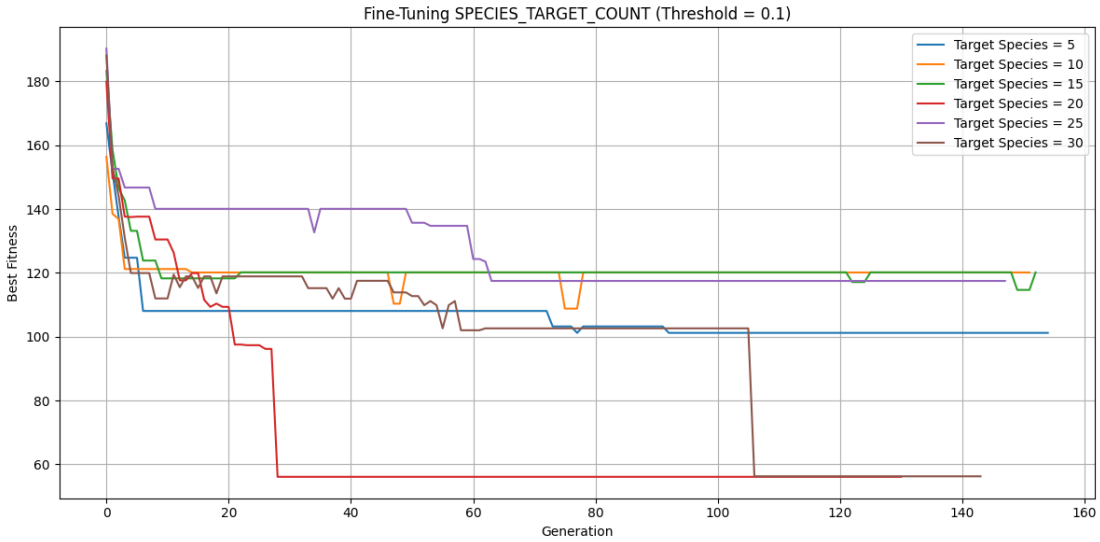
אך לקח זמן הרצה גדול ביותר מה שהשאיר אותנו בספק לגבי יעילות השימוש בו, לכן היה צורך בהשוואה שניה שהתבצעה בין שני הערכים תוך מתן פרק זמן גדול יותר להרצה:



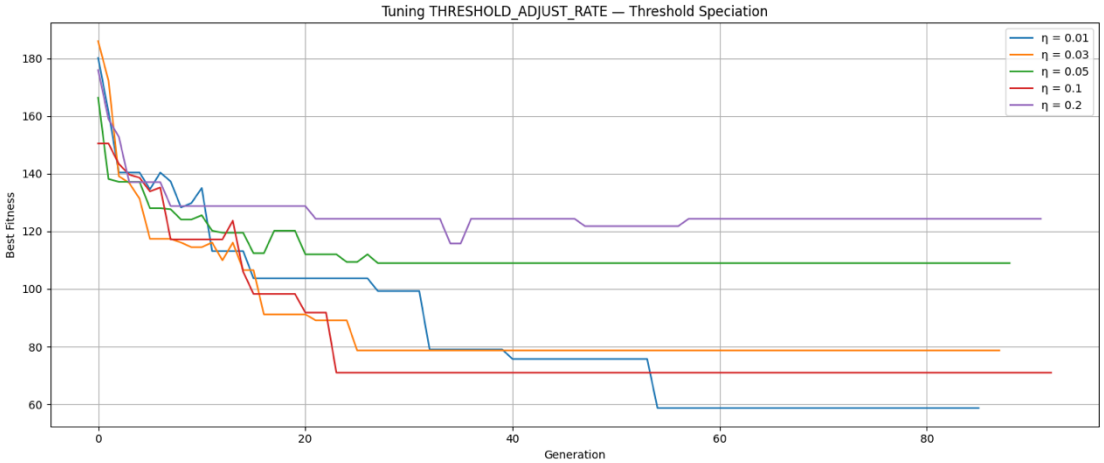
התוצאות מראות עדיפות ברורה ל- 0.1, ומההפרש הדומה בין מספר הדורות בשתי ההרצות נראה שההבדל בריצה לא יחסי אלא כנראה משתקך בשלבים הראשונים בלבד. אולם בגלל ש- 0.1 הוא ערך קצה (הכי קטן מבין אלה שנבדקו), והמרווחים בין הערכים שנסקרו גדול היה צורך בלבדוק ערכים בסביבתו גם כן ועל סמך כך נבחר 0.05 מכיוון שנתן את התוצאות הכי טובות.

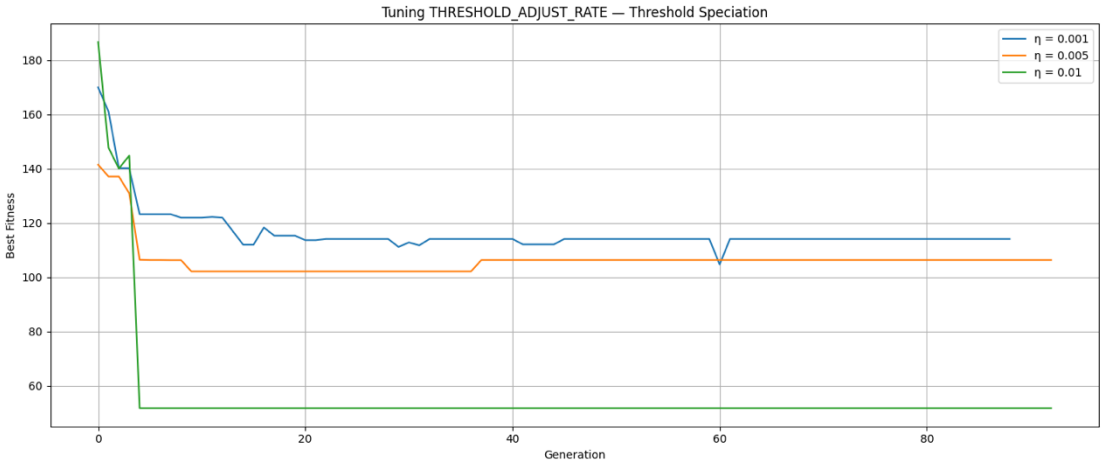


באשר למספר הזנים האידאלי, מבין המספרים 5-30 במרווחים של 5 התוצאות של 20 זנים היו הטובות ביותר לאורך רוב הדורות, במיוחד המתקדמים ועל כן הוא נבחר כערך של הפרמטר:

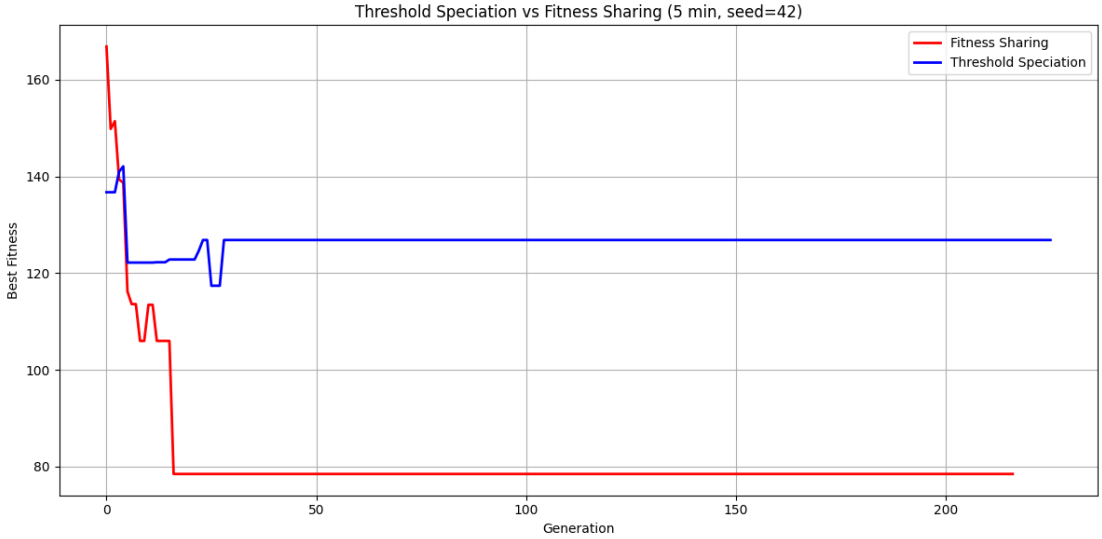


באשר למידת השינוי של הסף (Adjust Rate) התוצאות הצביעו על הערך של 0.1 כאפקטיבי ביותר:



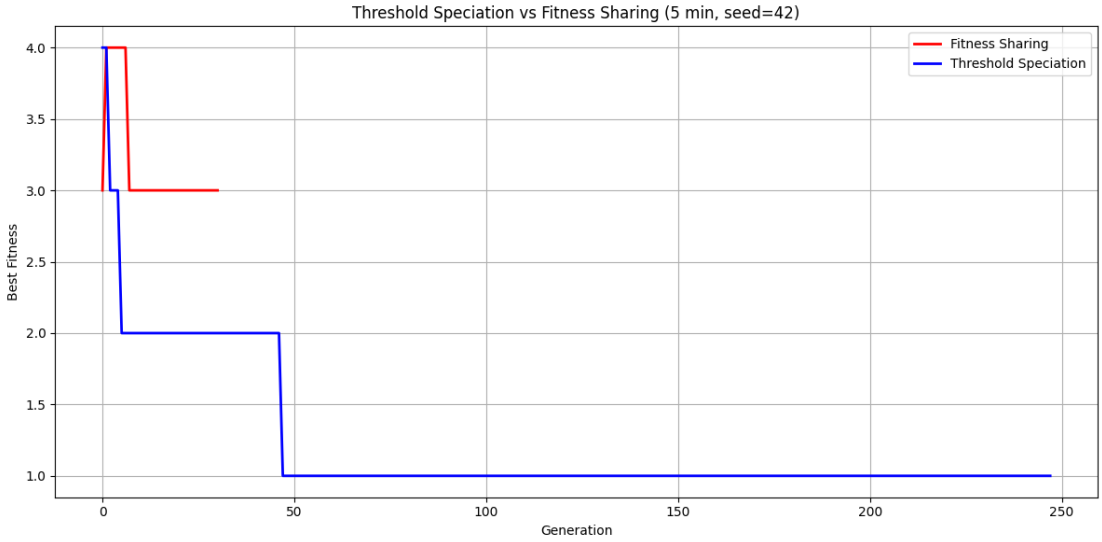


הביצועים של האלגוריתם על בעיית ה- DTSP בשימוש בשני האלגוריתמים תחת הקונפיגורציה הטובה ביותר שהתקבלה עבור כל אחד היו לטובת אלגוריתמים המחיצות באופן מכריע כפי שמציג הגרף להלן:



מהתוצאות נראה שהענשת הפרטים הדומים ועידוד הפרטים הייחודים מצליחה בלחלץ את האלגוריתם מהתכנסות לוקאלית באופן טוב יותר מאלגוריתם פיצול הזנים אשר משיג תוצאות טובות בהתחלה אולם ככל הנראה התייצבות המינים לאחר מכן מגבילה את השונות וההשתנות בתוך כל מין.

התוצאות עבור בעיית ה- Bin Packing היו באופן הפוך לטובת ה- Threshold Speciation. דבר שניתן לשים לב הוא הסיבוכיות הגדולה הכרוכה בשימוש באלגוריתם המחיצות עם הבעיה:



סעיף 5

ערכי הפרמטרים הטובים ביותר, שהיו דומים בשני המקרים של שתי הבעיות, על סמך מה התוצאות שהוצגו בסעיפים הקודמים היו באופן הבא:

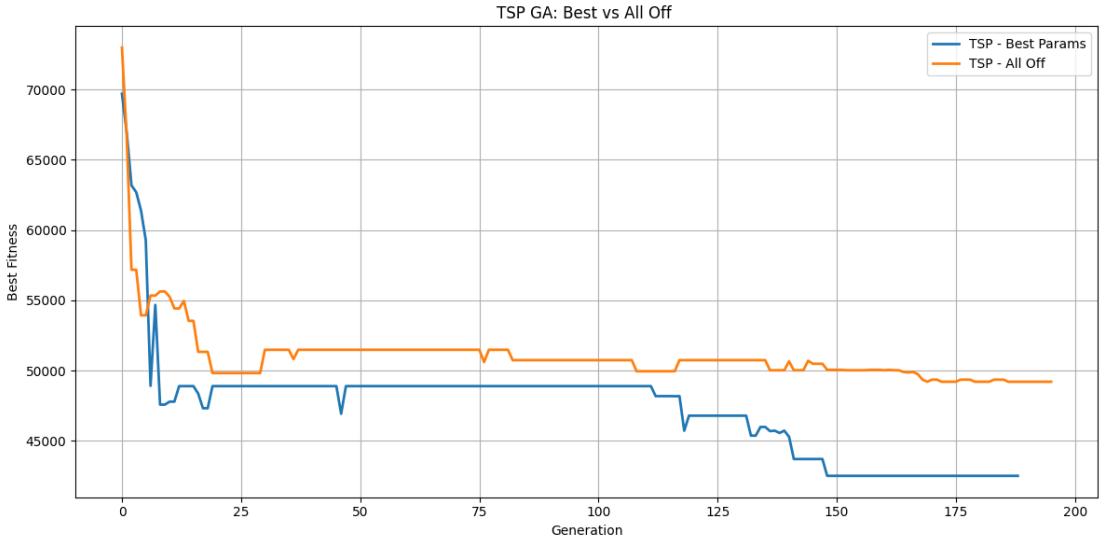
מדיניות השליטה בקצב המוטציות: Triggered Hyper mutation לפי ממוצע הפיטניס של האוכלוסייה.

שיטת המוטציות האינדיבידואלית: המבוססת פיטניס.

שיטת הפיטניס האינדיבידואלי: המבוססת גיל.

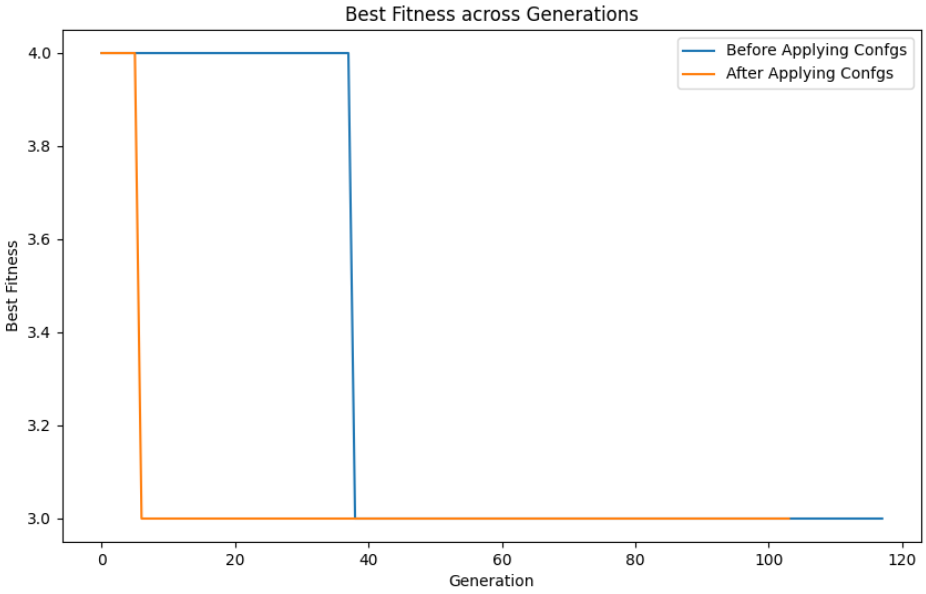
באשר ל- **Speciation**, בבעיית ה- DTSP, התקבלו תוצאות טובות יותר בעת השימוש באלגוריתם המחיצות, ואילו בבעיית ה- Bin Packing לא נראה שהשימוש באלגוריתמים שיפרה את התוצאות המצדיקה את הסיבוכיות הגדולה שהם הוסיפו.

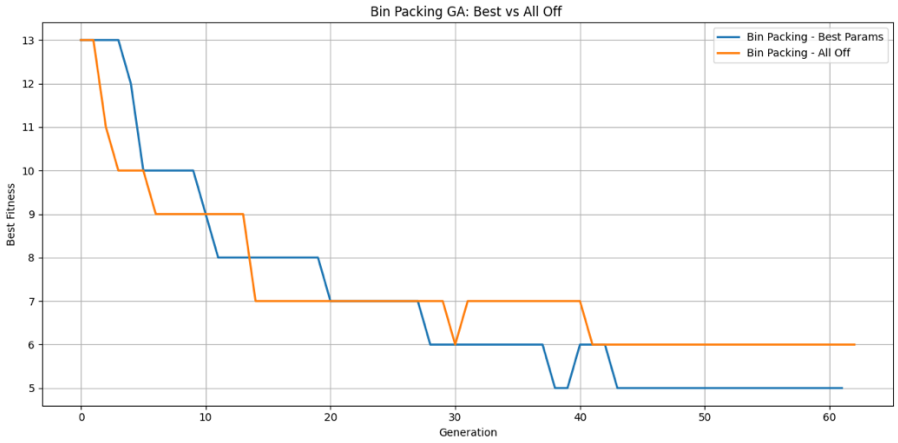
ההשוואה בין הביצועים לפני השיפורים שנעשו ואחרי בבעיית ה- DTSP היו כמוצג בגרף הבא כאשר הקו הכתום מייצג את הפיטנס הטוב ביותר לאורך הדורות בשימוש ללא השימוש בפרמטרים שהתווספו במהלך האלגוריתם, והכחול הוא עם האימוץ שלהם. בחרנו להשתמש בבעיה בעלת מימדים גדולים על מנת להבליט את ההבדל:



כבר בשלב מוקדם הקו הכחול פותח פער מול הכתום מה שמצביע על כך שהשיפורים אפשרו הגעה לפתרונות טובים יותר מהר מדי. כמו כן, השיפור היה עקבי ומתמשך, והפער התרחב ככל שהתקדמנו בדורות.

השוואה עבור בעיית ה- Bin Packing נעשתה תוך שימוש בשתי בעיות מכיוון שטווח המספרים של הראשונה לא אפשר להבדיל בין שני המצבים:





בדומה לבעיית ה- DTSP, השימוש בקונפיגורציה שהתקבלה הביא לשיפור קל גם במהירות ההתכנסות וגם באיכות הפתרון. בשל היות מרחב הפתרונות של בעיית ה- Bin Packing קטן יחסית השיפור נראה קטן בהתאם.

<u>סעיף 6</u>

מה המטרה של הניסוי ?

מטרת ניסוי זה היא לבחון את השפעתו של אפקט בולדווין על התנהגות של אוכלוסייה אבולוציונית. האפקט מתאר מצב שבו לומדים (כלומר פרטים שיכולים לשפר את ביצועיהם באמצעות תהליך למידה אישי – אך לא תורשתי) מצליחים לשרוד ולהשתפר, אף על פי שהשיפור הזה לא עובר לדור הבא ישירות. המטרה המרכזית הייתה לבדוק:

1. האם הלומדים תורמים לתהליך האבולוציוני?
2. כיצד משתווה רמת ההתאמה שלהם ללא-לומדים?
3. האם הלמידה מאפשרת שמירה על גיוון גנטי או מובילה להתכנסות טובה יותר?

הגדרת הבעיה:

במודל זה אנו עובדים עם **אוכלוסייה בגודל 1000 פרטים** ,כאשר כל פרט מיוצג על ידי גנום באורך 20 ביטים (0 או 1). המטרה של כל פרט היא **להתאים את הגנום שלו לגנום מטרה סודי** ,שנבחר באקראי בתחילת הסימולציה ואינו משתנה לאורך הדורות. האוכלוסייה מורכבת משני סוגי פרטים:

1. לומדים :מוגדרים עם הסתברות של 50% מהאוכלוסייה, כאשר כל לומד מקבל *קצב למידה* $r \epsilon [0.25,1]$ *הם אינם משנים את הגנום, אך ערך ההתאמה (fitness) שלהם מחושב כך שהוא מוכפל בקצב הלמידה, כלומר:* $r \cdot corrct_{bits} = fitness$
2. **לא לומדים** :פרטים רגילים, שיכולת ההתאמה שלהם שווה למספר הביטים הנכונים בגנום בלבד.

בתחילת כל דור, כל פרט:

1. נבחר להתרבות בהסתברות פרופורציונלית לערך ההתאמה שלו (Roulette Wheel Selection).
2. מעביר את הגנום לצאצא, כולל אפשרות למוטציה (flip של ביט) בהסתברות 1%.
3. הצאצא מקבל מחדש הגדרה אם הוא לומד או לא.

```python
# 2. Create initial population of individuals
def create_individual():
    genome = [random.randint(0, 1) for _ in range(GENOME_LENGTH)]
    is_learner = random.random() < LEARNING_PROPORTION

    learning_rate = random.uniform(*LEARNING_RATE_RANGE) if is_learner else None
    return {
        "genome": genome,
        "is_learner": is_learner,
        "learning_rate": learning_rate,
        "fitness": None,
        "correct": 0,
        "incorrect": 0
    }
```

בתמונה זו רואים את הפונקציה create_individual אשר אחראית ליצור פרט חדש לאוכלוסייה. כל פרט מורכב מגנום של 20 ביטים (אפסים ואחדים) בנוסף, נקבע אם הפרט הוא "לומד" או לא, לפי הסתברות מסוימת (LEARNING_PROPORTION).

הפרטים הלומדים יוכלו בהמשך "לשפר" את ביצועיהם בעזרת למידה – מה שמדמה את אפקט בולדווין.

חישוב הכשירות (Fitness) של כל פרט:

המטרה היא לחשב את הכשירות של כל פרט באוכלוסייה לפי רמת ההתאמה למטרת המטרה תוך התחשבות בשאלה אם הוא לומד או לא.

שלבי החישוב:

1. השוואה בין הפרט לבין הגנום הרצוי:

```
# Count correct bits
correct = sum([1 for i in range(GENOME_LENGTH) if genome[i] == goal[i]])
incorrect = GENOME_LENGTH - correct
```

- מחשב כמה ביטים תואמים למטרה correct וכמה לא incorrect.

2. קביעת ה FITNESS:

```
# Fitness calculation
if is_learner:
    fitness = correct * r
    learner_fitness.append(fitness)
else:
    fitness = correct
    non_learner_fitness.append(fitness)
```

- אם הפרט **לומד**, מקבלים עונש של r (מספר בין 0.25 ל 1) על מספר ההתאמות.
- אם הפרט **לא לומד**, הכשירות שלו היא פשוט מספר הביטים הנכונים.

3. שמירת הכשירות והסטטיסטיקות:

```
# Update individual
individual["fitness"] = fitness
individual["correct"] = correct
individual["incorrect"] = incorrect
```

- כל פרט שומר את ערך הכשירות שלו ואת מספר הביטים הנכונים והלא נכונים שלו.

4. ממוצעים לאוכלוסייה:

```
# Averages
avg_correct = total_correct / len(population)
avg_fitness_learners = sum(learner_fitness) / len(learner_fitness) if learner_fitness else 0
avg_fitness_nonlearners = sum(non_learner_fitness) / len(non_learner_fitness) if non_learner_fitness else 0
```

- ובסוף מחושבים ממוצעים לסטטיסטיקות לצורך הדמיה גרפית והשוואה.

הרעיון המרכזי של זה שפרטים לומדים נענשים (בצורה סימבולית) בגלל שהם מתאימים את עצמם כלומר, ההצלחה שלהם לא מגיעה מהתורשה. לעומת זאת, פרטים לא לומדים מקבלים ניקוד מלא אם הם מתאימים – ולכן אם נולדו "מושלמים", הם יתרבו מהר יותר. בצורה כזו נוכל לבדוק האם הלמידה מקדמת את האוכלוסייה לאורך הדורות.

בחירה ומוטציה (Selection + Mutation)

בחירת הורים מתבצעת באמצעות שיטת גלגל הרולטה (Roulette Wheel Selection) שבה סיכוי הבחירה של פרט עולה ככל שהכשירות שלו גבוהה יותר. אם אין כשירויות, נבחר פרט באקראי לחלוטין. **מוטציה** מתבצעת ביצירת הדור הבא: לכל גן (bit) יש סיכוי 1% לעבור הפיכה (flip) , כלומר מ־0 ל־1 או הפוך. זה מוסיף אקראיות ועוזר למנוע קיבעון של האוכלוסייה.

```python
def roulette_wheel_selection(population):
    total_fitness = sum(ind["fitness"] for ind in population)
    if total_fitness == 0:
        # fallback: uniform random selection
        return random.choice(population)

    pick = random.uniform(0, total_fitness)
    current = 0
    for ind in population:
        current += ind["fitness"]
        if current >= pick:
            return ind
    return population[-1]  # fallback
```

הדור הבא נוצר בפונקציה reproduce. כל פרט חדש נוצר ע"י שכפול של פרט נבחר, יחד עם יישום של מוטציה ושיוך מחדש של תכונת "לומד" (כולל רמת הלמידה r אם נבחר כלומד).

```python
def reproduce(population):
    new_population = []

    for _ in range(POP_SIZE):
        parent = roulette_wheel_selection(population)
        parent_genome = parent["genome"]

        # Copy genome and apply mutation (bit flip)
        child_genome = [
            bit if random.random() > MUTATION_RATE else 1 - bit
            for bit in parent_genome
        ]

        # Assign learner status and learning rate fresh
        is_learner = random.random() < LEARNING_PROPORTION
        learning_rate = random.uniform(*LEARNING_RATE_RANGE) if is_learner else None

        child = {
            "genome": child_genome,
            "is_learner": is_learner,
            "learning_rate": learning_rate,
            "fitness": None,
            "correct": 0,
            "incorrect": 0
        }

        new_population.append(child)

    return new_population
```

- לכל ביט בגנום של הצאצא יש סיכוי של 1% לעבור *היפוך*.
- *בעזרת זה מוטציה מוסיפה אקראיות ומאפשרת לאלגוריתם להימנע מקיבעון ולחקור פתרונות חדשים.*
- כל פרט חדש נבחר באקראי האם הוא לומד או לא. אם הוא לומד, מוקצה לו ערך r אקראי.

לולאת האבולוציה (Evolution Loop):

בשלב זה מתבצעת הליבה של תהליך הלמידה האבולוציונית — עבור כל דור:

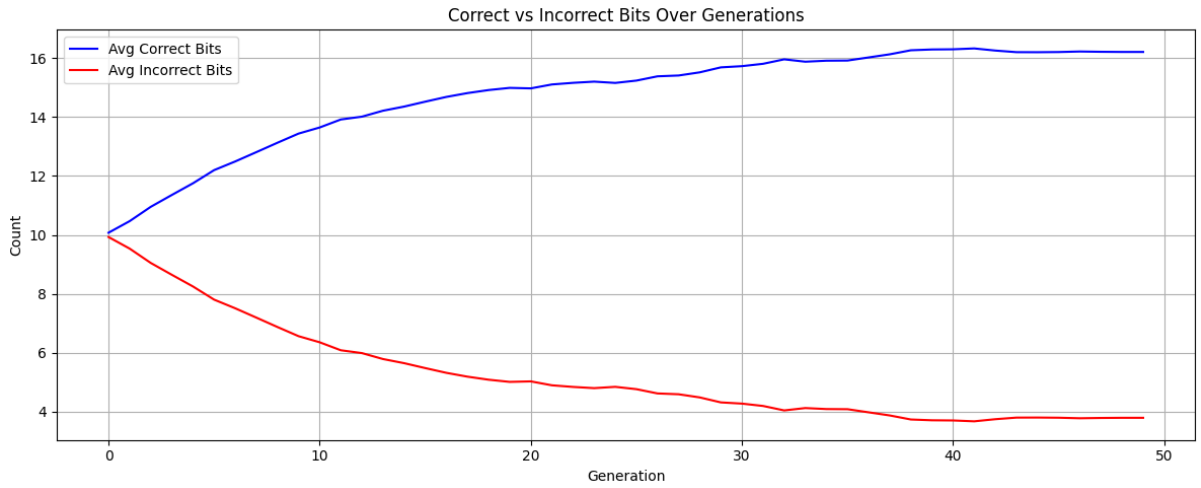
1. חישוב הכשירויות של כלל האוכלוסייה לפי מידת ההתאמה של כל פרט למטרה, עם התייחסות לשאלה האם הוא לומד או לא.
2. שמירת ערכי סטטיסטיקה: ממוצע של מספר הביטים הנכונים, וממוצעי הכשירויות של לומדים ולא-לומדים (כפי שהוזכר בסעיפים הקודמים).
3. בדיקת תנאי עצירה:
   - אם התקבלה התאמה מלאה.(Fitness = 0)
   - אם לא חל שיפור במשך 50 דורות רצופים.
   - אם עבר זמן הריצה המוגדר.
4. יצירת דור חדש: באמצעות הפונקציה reproduce

```python
# Start evolution loop
for generation in range(NUM_GENERATIONS):
    avg_correct, avg_learners, avg_nonlearners = evaluate_fitness(population, goal_genome)

    # Track stats
    avg_correct_list.append(avg_correct)
    avg_fitness_learners_list.append(avg_learners)
    avg_fitness_nonlearners_list.append(avg_nonlearners)

    print(f"Gen {generation:2}: Avg Correct = {avg_correct:.2f}, Learners = {avg_learners:.2f}, Non-Learners = {avg_nonlearners:.2f}")

    # Create next generation
    population = reproduce(population)
```

גרפים ומסקנות

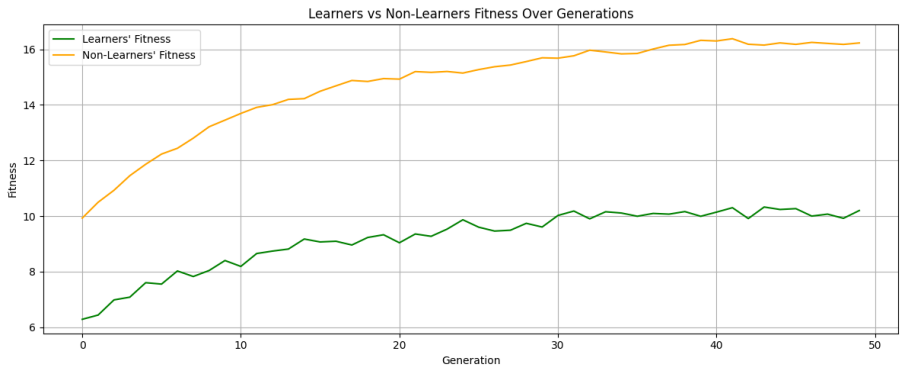
לאחר ריצה של 50 דורות, הפקנו שני גרפים עיקריים:

1. הגרף הראשון מציג את מספר הביטים הנכונים והלא-נכונים לאורך הדורות. ניתן לראות מגמה ברורה של עלייה במספר הביטים הנכונים, מה שמעיד על כך שהאוכלוסייה לומדת ומשתפרת.



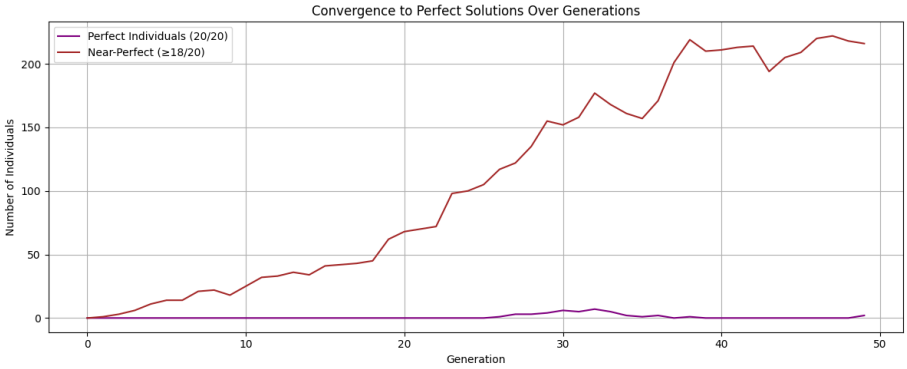
ככל שהדורות מתקדמים, מספר הביטים הנכונים הממוצע (בכחול) באוכלוסייה עולה בהתמדה מ 10 ביטים בדור הראשון ועד כ 16.2 ביטים בדור 50.

2. הגרף השני מציג את רמת הכשירות של לומדים מול לא-לומדים לאורך הדורות. ניתן לראות בבירור כי רמת הכשירות של הלא-לומדים גבוהה יותר כמעט לכל אורך הריצה.



כבר מהדור הראשון, הלא-לומדים מצליחים להשיג כשירות גבוהה יותר (כמעט 10) לעומת הלומדים שמתחילים בערך 6.2. גם בהמשך, למרות עלייה הדרגתית בכשירות הלומדים, הם אינם משיגים את הלא-לומדים. הסיבה לכך נעוצה בכך שכאשר פרט לומד, הכשירות שלו נענשת על ידי מקדם הלמידה r, לכן לרוב תהיה נמוכה יותר. מצד שני, לא-לומדים שמתחילים עם התאמה גבוהה ישמרו עליה ואף יעברו הלאה לדורות הבאים.

3. הגרף השלישי מציג את מספר הפרטים שהתקרבו לפתרון המושלם – כלומר כאלו שהתאימו לחלוטין (20/20) או כמעט לחלוטין (לפחות 18/20) לגנום המטרה.



בדור הראשון אין כמעט פתרונות טובים, אך לאחר כ־20 דורות מתחילים לראות מגמת עלייה חדה בפרטים שהתקרבו לפתרון. בסביבות דור 35 נראית קפיצה דרמטית כאשר כ 200 פרטים מתוך 1000 מגיעים ל 18 ביטים נכונים או יותר. עם זאת, מספר הפתרונות המושלמים (20/20) נותר נמוך. המשמעות היא שהאוכלוסייה מתכנסת לאזורים טובים במרחב הפתרונות, אך עדיין שומרת על מידה של גיוון.

מסקנות כלליות מהסימולציה – האם אפקט בולדווין נצפה?

לאורך הסימולציה ניתן לראות מספר תופעות שמעידות על קיומו של אפקט בולדווין:

1. השפעת הלומדים על האבולוציה :למרות שלומדים מתחילים עם ערך כשירות נמוך יותר, הם מצליחים להוביל את האוכלוסייה להתקדמות עקבית. זה בא לידי ביטוי בעלייה במספר הביטים הנכונים לאורך הדורות.
2. תועלת עקיפה ללומדים :הלמידה עוזרת לפרטים "לשרוד" גם אם הם לא מותאמים לחלוטין, ולכן מאפשרת להם להשתתף בתהליך האבולוציוני – ובכך לקדם גנומים שיכולים להשתפר בדורות הבאים.
3. הצטיינות לא-לומדים :הפרטים הלא-לומדים שמתחילים עם התאמה גבוהה מקבלים יתרון, אך ללא תמיכה של לומדים, הסיכוי להגיע לרמות גבוהות של דיוק עבור כלל האוכלוסייה היה קטן יותר.
4. התכנסות הדרגתית :ניתן לראות עלייה במספר הפתרונות הקרובים למושלם (גרף 3) שמתחילה רק לאחר כמה עשרות דורות – דבר שמרמז שהלמידה תומכת בגיוון ויציבות אבולוציונית.

לפיכך, ניתן לקבוע שהאפקט אכן נצפה בסימולציה שלנו, והוא מתבטא בהשפעה חיובית של הלומדים על קצב ההתקדמות האבולוציוני גם אם הם עצמם לא "מצליחים" על פני הלא-לומדים.