דו"ח מעבדה מס' 3

תאריך הגשה :

01.05.2020

מגישים:

דניאל שלם – 205745052

אלכסיי אלייב - 322162330

לפתרון בעיית Bin packing השתמשנו בFitness function הבאה:

$$F = \frac{\sum_{i=1}^{n} \left(\frac{f_i}{c}\right)^k}{n}$$

כאשר fi מסמל את המשקל של הבין הi, C את המגבלה של כל תא, n את מספר התאים בשימוש Ki הוא פרמטר שהתפקיד שלו הוא שליטה במשקלים בתאים.

Cross-over – השתמשנו בגרסה שונה של ordered crossover, ניקח חצי מהערכים של ההורה הראשון, ובהכנסה של ההורה השני תחילה ננסה במקום המתאים, אם משקל השק עובר את המגבלה המותרת נכניס לשק רנדומלי אחר.

האתחול של כל גן נעשה ע"י בחירת תא רנדומלי לכל חפץ.

מבנה הנתונים בו השתמשנו הוא רשימה בגודל N שכל אינדקס מייצג חפץ וכל ערך מייצג את התא בו החפץ נמצא.

***כל הפתרונות הם שלמים ואין חריגה של מגבלת המשקל.**

השלמות נובעת מכך שכל פעם שנמצא את הגן הטוב ביותר, נעבור על כל הגנים הממוינים עד שנמצא גן בו אין תא עם משקל העובר את המגבלה.

120 חפצים: (K=2)

מס' איטרציות – 200

גודל אוכלוסייה - 500

תוצאות:

```
Average fitness: 5.342723342723006
Standard deviation:  0.37210747732720995

Number of empty bins 49

Generation running time for iteration 199:  0.08674907684326172
Absolute running time:  17.61486291885376
```

250 חפצים: (K=2)

מס' איטרציות – 200

גודל אוכלוסייה – 500

```
Average fitness: 5.3435214481730045
Standard deviation:  0.3580730640128328

Number of empty bins 100

Generation running time for iteration 199:  0.1655564308166504
Absolute running time:  34.468862533569336
```

500 חפצים: (K=2)

מס' איטרציות – 200

גודל אוכלוסייה - 500

```
Average fitness: 5.402220465385314
Standard deviation:  0.37288445008129933

Number of empty bins 219

Generation running time for iteration 199:  0.3447427749633789
Absolute running time:  76.44780015945435
```

1000 חפצים: (K=2)

מס' איטרציות – 200

גודל אוכלוסייה – 500

```
Average fitness: 5.320600523100271
Standard deviation:  0.3535142315171352

Number of empty bins 375

Generation running time for iteration 199:  0.6821920871734619
Absolute running time:  141.49480319023132
```

פונקציות מרחק:

לבעיית בול הפגיעה מימשנו את מרחק לוינשטין,

לבעיית N Queens מימשנו את מרחק קנדל-טאו בעזרת מימוש מהיר במיוחד שמצאנו במאמר באינטרנט (המאמר נמצא בקוד),

לבעיות Knap sack ו Bin packing השתמשנו במרחק אוקלידי שמשמש לייצוג ווקטורי.

Fitness Sharing:

```
@staticmethod
def sharing_function(distance, sigma_share, alpha):
    return 1 - (distance / sigma_share) ** alpha if distance < sigma_share else 0
```

לבעיות **מקסימיזציה חילקנו** את הfitness בshare fitness ולבעיות **מינימיזציה הכפלנו**.

הפתרון בעזרת הבעיה הזו מעלה את זמן הפתרון **משמעותית** וגם את הגיוון באוכלוסייה.

כנראה ששימוש בMulti-threading היא עוזר מאוד, אך לא הספקנו מבחינת זמנים.

בדוגמא הבאה ניתן לראות כי לעומת הפתרונות הקודמים שם חווינו התכנסות מהירה וסטיית התקן הייתה סביב 0.3 כאן סטיית התקן גדולה מ1 ברוב המקרים וגם איכות הפתרון טובה יותר (לא משמעותית).

כל המרחקים חולקו בגודל האובייקט על מנת להוריד את טווח המרחקים.

SIGMA SHARE = 4.5

```
Generation running time for iteration 48:  4.090576887130737
Sharing fitness
Share fitness time:  3.5938878059387207
Best:  [33, 95, 16, 31, 36, 58, 81, 46, 94, 97, 60, 71, 25, 86,
Average fitness: 0.8276485292946268
Standard deviation:  1.2861638124866144

Number of empty bins 55
```

לאחר 106 איטרציות:

```
Generation running time for iteration 106:  3.6315255165100098
Sharing fitness
Share fitness time:  3.5699942111968994
Best:  [64, 91, 117, 111, 48, 112, 35, 27, 44, 42, 109, 73, 26, 105,
Average fitness: 2.3858980043007594
Standard deviation:  3.368889046358552

Number of empty bins 53
```

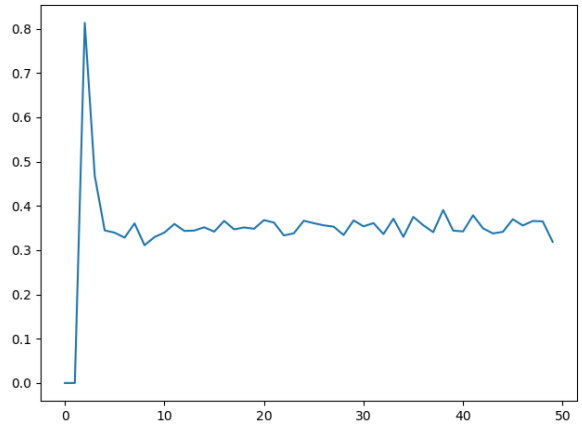
אפשר לראות שסטיית התקן ממשיכה לעלות ככל שמתקדמים באיטרציות, וזה מצביע על גיוון רחב של מרחב הפתרונות.

```
Share fitness time:  3.5829954147338867
Best:  [38, 51, 52, 102, 56, 80, 10, 57, 5, 12, 110, 29, 91, 103,
Average fitness: 12.607236905758574
Standard deviation:  15.223211844335218

Number of empty bins 51

Generation running time for iteration 199:  3.6727144718170166
```

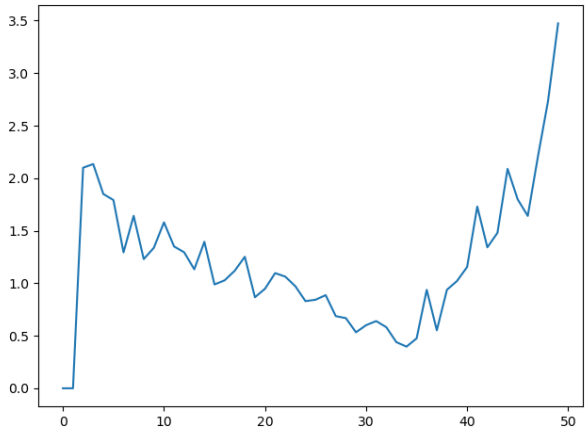
כלומר הפתרון שנמצא הוא פחות טוב מפתרונות קודמים שנמצאו גם ללא הshare fitness בגלל הבעיה אותה שי הזכיר בשיעור שהיא שShare fitness עלולה לפגוע בפתרונות האיכותיים במטרה לחפש הרבה פתרונות.

השתנות סטיית התקן לאורך כל איטרציה:

ללא Share fitness

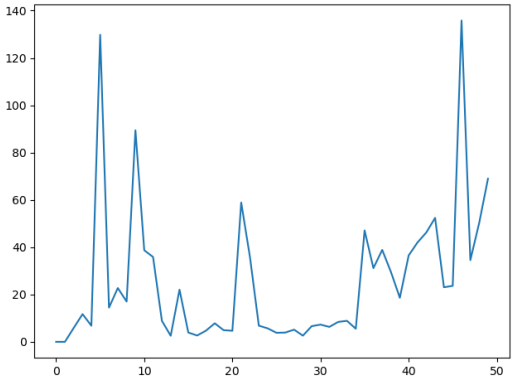


עם Share fitness



ניתן לראות שסטיית משתנה משמעותית בעזרת השימוש בShare fitness, אך הסיבוכיות והקושי בבחירת הSigma share לכל בעיה מקשים את השימוש בשיטה זו.

בעיית דוגמא 2:




```
Number of empty bins 107
```

בגרף זה ניתן לראות איך השיטה מצליחה להוציא את עצמה ממינימום מקומי כל פעם מחדש, ולבסוף להביא פתרון מצוין.

פתרון שמצאנו בשביל sigma share הוא לבצע Scaling על מטריצת הדמיון של האוכלוסייה, אבל הסיבוכיות שוב עולה ולמרחבי פתרון גדולים זה בעייתי.

Threshold Speciation:

חלוקה של האוכלוסייה לזנים שונים, למימוש שיטה זו נעבור על כל גן ונמצא את הזן הראשון שמתאים לו. אם לא קיים זן כזה, ניצור אחד חדש. המטרה היא שמס' הזנים יהיה כמה שיותר קרוב למס' הזנים האופטימלי (30).

במהלך הcrossover לא ניתן לבצע crossover בין שתי גנים מאותו זן, אלא אם קיים זן אחד באוכלוסייה או שחיפשנו זן שונה מס' מוגבל של איטרציות.

לאחר כל התאמה של הגנים לזנים, ביצענו התאמה של פרמטר הThreshold כנדרש:

```
def adjust_speciation_threshold(self, max_iterations=3):
    """Adjust the number of species to be around max_species"""
    iterations = 0

    while len(
            self.species_list) != self.optimal_species and self.speciation_threshold > 0 and iterations < max_iterations:
        iterations += 1
        self.speciation_threshold += 1 * np.sign(len(self.species_list) - self.optimal_species)/(iterations+2)
        self.speciation_threshold = max(1, self.speciation_threshold)
        self.species_list = self.make_species(adjust_threshold=False)
```

השיטה שלנו להתאמת פרמטר הסף היא, לעבור 3 איטרציות במקסימום על בניית הזנים מהאוכלוסייה, ואז להחסיר מפרמטר הסף את ההפרש ביניהם (חיובי או שלילי, בהתאם) חלקי האיטרציה הנוכחית. כלומר, בכל שלב של העדכון נחלק במס' גדול יותר ולכן השינוי יהיה קטן יותר.

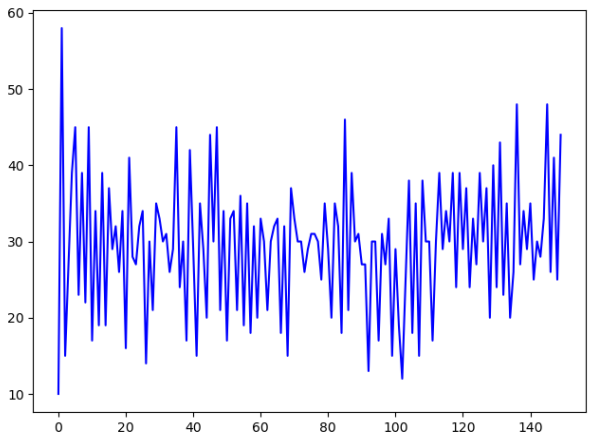
תופעה מעניינת שקשורה לשיטה זו היא שאתחול נכון של פרמטר הסף עוזר מאוד למהירות הפתרון.

תופעה נוספת ששמנו לב אליה היא ששימור מס' זנים קבוע פוגע בשלמות הפתרון כנראה מכיוון ששומרים על מרחק בין הזנים.

<u>הגרפים הבאים יציגו את השינו במס' הזנים לאורך 150 איטרציות, את זמן הפתרון ואת מס' הבינים הריקים הגדול ביותר.</u>

בעיית דוגמא 1:

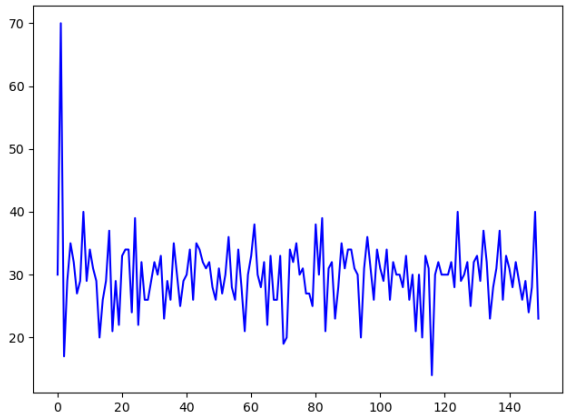
גרף המתאר את השינוי בבמות הזנים (אתחול לא נכון) :




```
Number of empty bins for best individual is: 66
Generation running time for iteration 149:  0.1137
Absolute running time:  35.48108649253845
```

בעיית דוגמא 2:

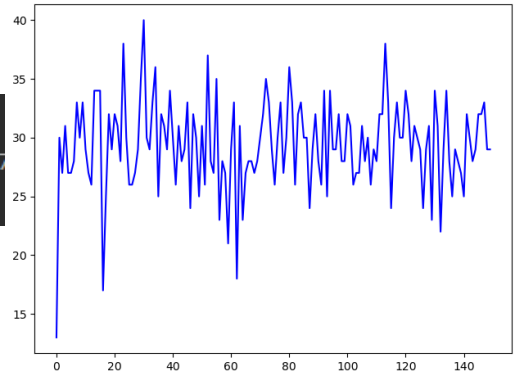
```
Number of empty bins for best individual is: 135
Absolute running time:  52.49428844451904
```



ניתן לראות איך האלגוריתם מנסה להתמודד עם השינויים באוכלוסייה ותמיד לנסות להגיד שוב לכמות הזנים האופטימלית. <u>הממוצע נע סביב ה30 זנים, כנדרש.</u>

בעיית דוגמא 3:

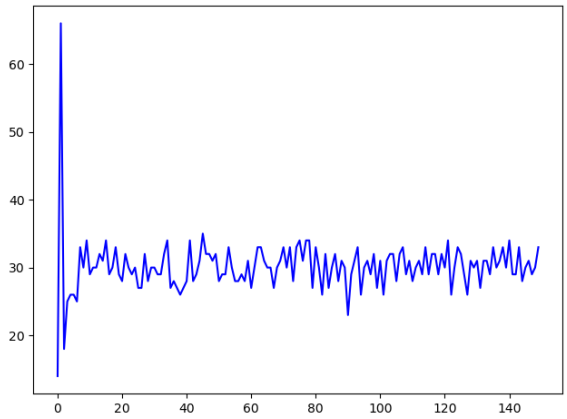
```
Number of empty bins for best individual is: 245
Generation running time for iteration 149:  0.38596725463867
Absolute running time:  94.03738737106323
```



שוב, הוא חוזר ומגיע לאזור ה30 למרות אתחול לא טוב. גם פה התוצאות הן הטובות ביותר.

בעיית דוגמא 4:

```
Number of empty bins for best individual is: 524
Best:  [189, 635, 537, 896, 565, 155, 858, 954, 225, 41,
Average fitness: 0.5513459729753387
Standard deviation:  0.08772253383601705

Generation running time for iteration 149:  0.8028514385
Absolute running time:  153.77762603759766
```



בבעיות 3,4 זמן הריצה הוא ארוך, ניתן לבצע את הפעולות עם זמן ריצה קצר יותר ע"י הורדת גודל האוכלוסייה או מס' האיטרציות.

Species היא לבינתיים השיטה המועדפת עלינו, מכיוון שהפרמטר בה הוא דינאמי ומשתנה במהלך האלגוריתם, וגם כי התוצאות שלה הן הטובות ביותר (קרובות מאוד לאופטימליות) לאחר מס' קצר מאוד של איטרציות.

מימוש השיטה באמצעות אלגוריתם Clustering כנראה ייתן את הביצועים הטובים ביותר.

Local optima:

לזיהוי Local optima השתמשנו בשני סיגנלים כפי ששי הסביר בביתה.

1. **סטיית תקן שואפת ל0 –** ביצענו זאת באמצעות דגימה של סטיית התקן של ה3 דורות האחרונים, חישוב הממוצע שלהם ובדיקה האם הוא מתחת לסף מסוים, ובדיקה האם סטיית התקן יורדת לאורך הדורות (שואפת ל0).
2. **דמיון באוכלוסייה –** ביצענו זאת ע"י דגימה אקראית של 10% מהאוכלוסייה, חישוב המרחקים ביניהם, חישוב ממוצע המרחקים וכמו בסטיית התקן, לבד האם המרחקים שלהם מתחת לסף מסוים. **בנוסף,** ביצענו scaling של המרחק שקיבלנו ל[0,2].

```
# Checking std sign for local optima
if np.mean(generation_std[-local_optima_range:-1]) <= std_threshold:
    std_signal = True

# Checking similarity sign for local optima
if current_population.calc_similarity() <= similarity_threshold:
    similarity_signal = True
```

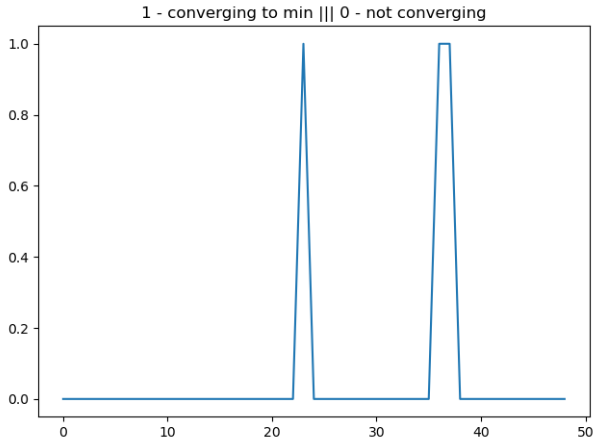
טיפול בLocal optima –

בחרנו לממש את שיטת **Hyper mutation** על מנת להיחלץ מlocal optima.

את השיטה מימשנו בצורה הבאה:

שמרנו לאורך כל דור מערך של flags של local optima, אם אנחנו נמצאים בlocal optima לאורך 3 דורות, נפעיל את Hyper mutation שתעלה את הסתברות **לmutation ל0.7 ואת מס' הmutation לאורך הגן חלקי ,20 או 5, הגדול מבניהם**. כלומר, לא רק שקצב המוטציות יעלה, אלא גם מס' המוטציות שקורות.

אם אנחנו רק נכנסים לlocal optima, קצב הmutation יעלה ל0.5 ומס' הmutation יעלה ל3.

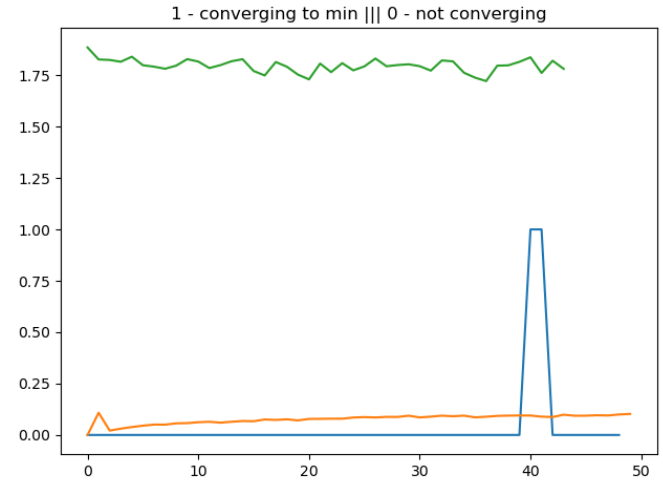


בגרף הנ"ל ניתן לראות איך הhyper mutation מוציאה את הפתרון ממינימום לוקאלי כאשר הוא מתכנס לשם.

דוגמא נוספת ניתן לראות איך הפתרון מתכנס למינימום לוקאלי ואנחנו יוצאים ממנו בעזרת Hyper mutation :

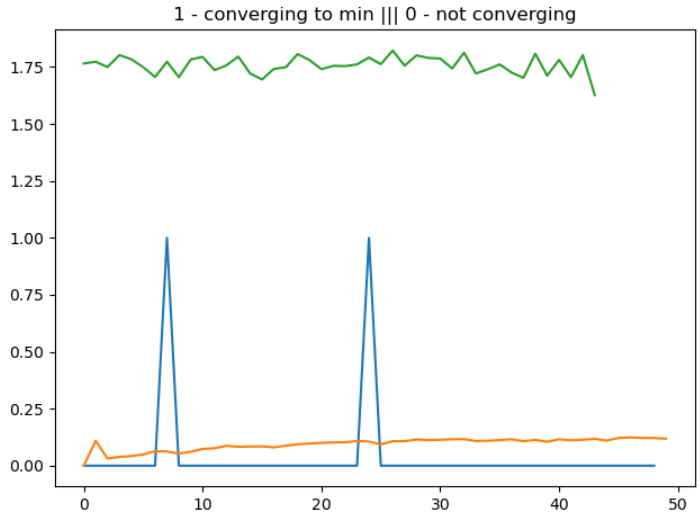
<u>ללא Hyper mutation:</u>

צבע ירוק – מרחק, צבע כחול – התכנסות למינימום, צבע כתום - STD



עם Hyper mutation:

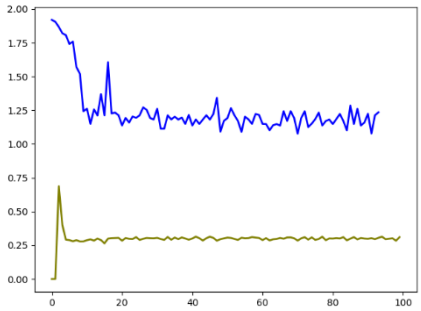
צבע ירוק – מרחק, צבע כחול – התכנסות למינימום, צבע כתום - STD



ניתן לראות כי הבריחה היא מיידית בעזרת הHyper mutation.

Hyper Mutation + Species:

בעיה מס' 4:




```
Number of empty bins 464
```

לדעתנו השילוב הטוב ביותר הוא שילוב של שניהם מכיוון שהHyper mutation נותנת משקל גם לדמיון בין הגנים וגם לממוצע סטיית התקן.

NSGA-2

```python
# First function to optimize
def function1(x_vector):
    y = 0
    for x in x_vector:
        y -= x ** 2
    return y
```

```python
# Second function to optimize
def function2(x_vector):
    y = 0
    for x in x_vector:
        y -= (x - 2) ** 2
    return y
```

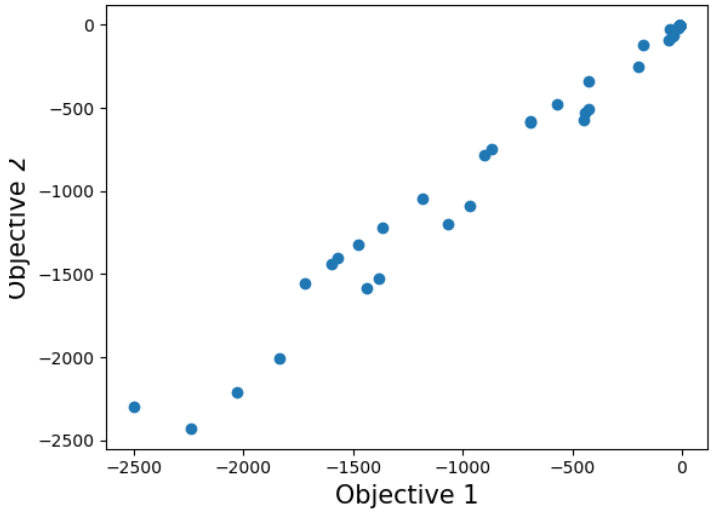
TwoPointCrossover – Crossover

על מנת ליצור את הצאצאים הרצנו בכל שלב טורניר שהמנצח בו הוא זה שהRank שלו הוא הגבוהה ביותר, או שאם יש כמה זהים אז זה בעל ה Crowding distance הגדול ביותר ינצח.

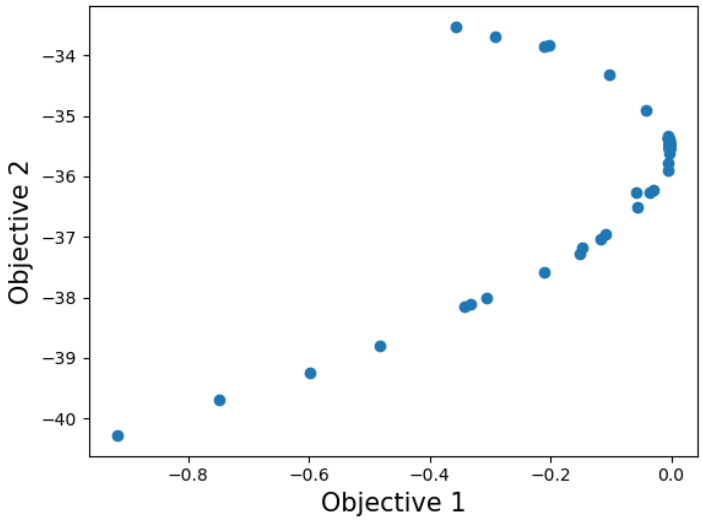
ה Rank של כל מתמודד נקבע בעזרת non dominated sorting ששם גם נקבעת החזית המתאימה, המרחק נקבע בעזרת Crowding distance.

חזית פארטו מינימלית דוגמאות:

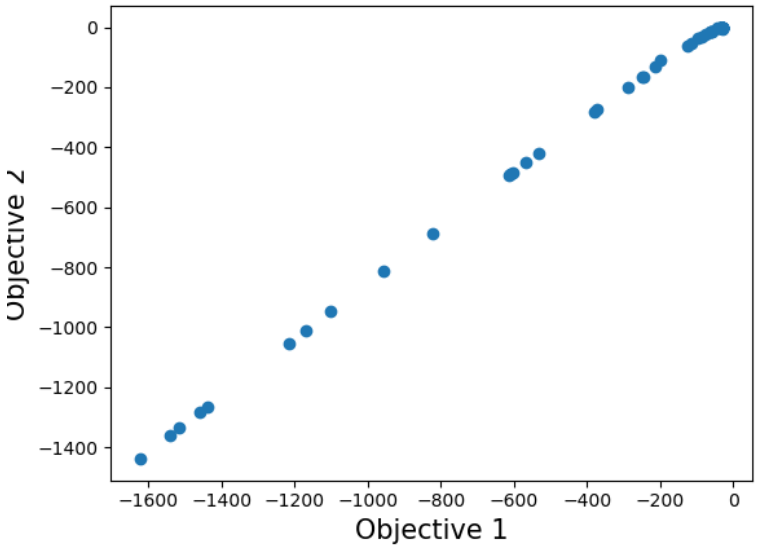
x1 = -50 ; x2 = 50 ; n = 5



x1 = -0.99 ; x2 = 0.99 ; n = 8



x1 = 0 ; x2 = 40 ; n = 10



Baldwin Effect

את הניסוי אתחלנו כנדרש.

```
unknown_ind = idx_list[0:int(N / 2)]  # Half indices for '?'
true_ind = idx_list[int(N / 2):int(N / 2) + int(N / 4)]  # Quarter will be wrong
false_ind = idx_list[int(N / 2) + int(N / 4):]  # Quarter will be right
```

את החיפוש הלוקאלי נבצע בחישוב הFitness function כנדרש.

כאמור, 25% מהביטים יהיו נכונים, 25% לא נכונים, 50% רנדומליים ('?').

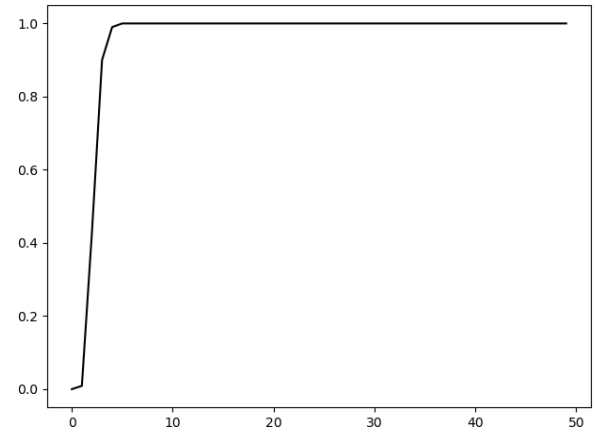
האלגוריתם הממטי שלנו יוצר בצורה הבאה:

כל פעם שנרצה להתחיל local search, נבדוק את מס' הביטים הנכונים, ומס' הביטים הלא נכונים. במידה ומס' הביטים הנכונים גדול ביותר מ2 ממס' הביטים הלא נכונים נבצע חיפוש, אם לא, לא נבצע.

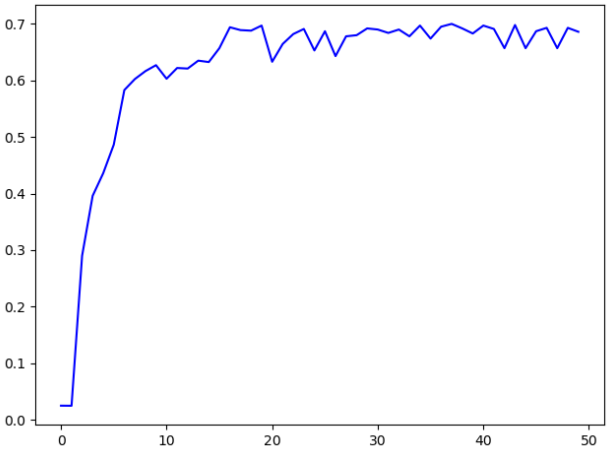
הוספת השיטה התגלה כיעילה מאוד מבחינת זמן ומבחינת איכות הפתרון.

תוצאות הניסוי נצפו גם אצלנו:

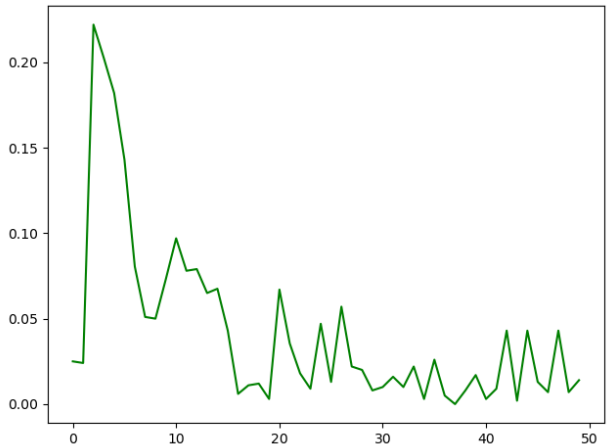
גרף אחוז הלומדים בכל דור:



גרף המיקומים הנכונים:

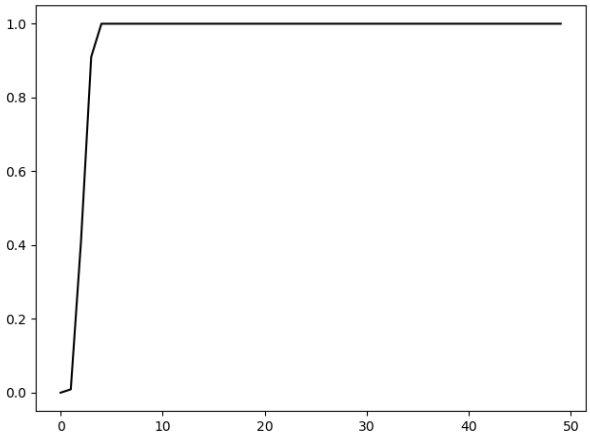


גרף המיקומים הלא נכונים:

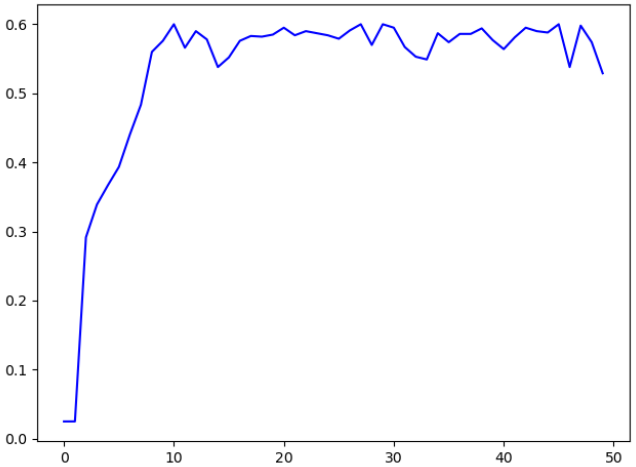


הרצה נוספת:

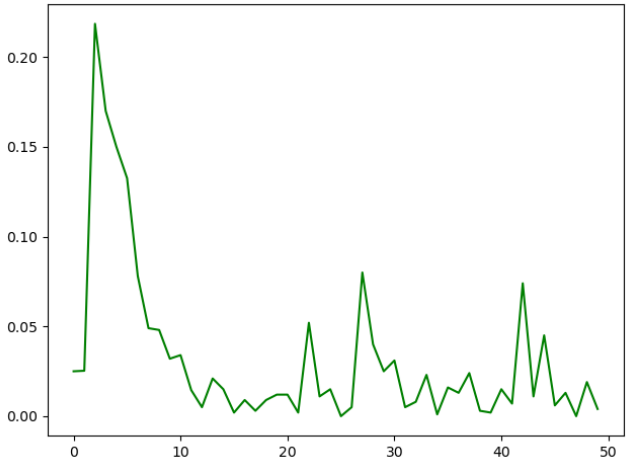
גרף אחוז הלומדים בכל דור:



גרף המיקומים הנכונים:



גרף המיקומים הלא נכונים:



מהתוצאות האלו ניתן להסיק כי למידה אכן עוזרת, אך צריכים להשתמש בה ביעילות בגלל העלאת הסיבוכיות הנגמרת כתוצאה מהחיפוש.