מעבדה 2 בבינה מלאכותית

נושא:

אלגוריתמים גנטיים, ממטיים ואבולוציה

מגישים:

א) אילן גודיק, 316315332

ב) יובל אלפסי, 318401015

מנחה:

מר שי בושינסקי

תאריך הגשה:

8 לאפריל 2016

תוכן עיניינים

חלק א: שכלול המנוע הגנטי

שאלה 1 – עמוד _ – דיווח זמן ריצה

שאלה 2 – עמוד _ – שיטות בחירה ושיטות שרידות

שאלה 3 – עמוד _ – פונקציית מרחק בין גנים

שאלה 4 – עמוד _ – אבחון מינימום מקומי

שאלה 5 – עמוד _ – היחלצות ממינימום מקומי

שאלה 6 – עמוד _ – בחינת ההשפעה על האלגוריתם הגנטי

שאלה 7 – עמוד _ – חזית פרטו אופטימל

חלק ב: אפקט בולדווין

חלק ג': ניתוחים וסטטיסטיקה

סדר גודל העבודה:

- מעבדה 1: 45 שעות
- מעבדה 2: 55 שעות
- כמו שבועיים וחצי של Full Time Job מתוך חודש לימודים אחד.
- 3750 שורות קוד
- ~150 commits

המנוע הגנטי

ארכיטקטורה

תחילה נתאר את מבנה המנוע הגנטי החדש, תהליך האבולוציה והרכיבים הכלולים בכך.

הפרדה: המנוע הגנטי מופרד לחלוטין מייצוג הבעיה והתכונות שלה, כך שניתן לבחור ולהחליף את המנוע ואת הבעיה באופן בלתי תלוי בזמן הריצה.

Generation:

זהו המרכיב הכולל המרכזי במנוע הגנטי, והוא מתאר כיצד אנחנו עוברים מGeneration אחד לGeneration הבא, וזה כולל:

```
public class Generation {
    public final ParentSelection selection;
    public final MutationStrategy mutationStrategy;
    public final SurvivorSelection survivorSelection;
    public final FitnessMapping[] fitnessMappings;
}
```

1. אסטרטגית בחירת הורים
2. אסטרטגית מוטציה
3. אופן בחירת השורדים לדור הבא בהתבסס על האוכלוסיה הקודמת וStream אינסופי של ילדים
4. אוסף פונקציות מיפוי לFitness, שמופעלות על כל הגנים באוכלוסיה.

המנוע הגנטי:

```
case class GeneticEngine(localOptimaSignal: LocalOptimaSignal,
                         normalGeneration: Generation,
                         localOptimaGeneration: Generation,
                         PopulationSize: Int)
```

מנוע גנטי מכיל:

1. אלגוריתם זיהוי מינימום לוקאלי
2. Generation כאשר האוכלוסיה היא לא במינימום לוקאלי
3. Generation כאשר האוכלוסיה היא כן במינימום לוקאלי

השיטה שלנו למידול התמודדות עם מינימום לוקאלי היא ע"י הפרדת כל הפעולות הגנטיות שנעשות כאשר האוכלוסיה לא במינימום לוקאלי לעומת מתי שהיא כן במינימום לוקאלי, וזה כולל אוסף פרמטרים שונה לכל שיטה וגם פעולות גנטיות שונות כגון Random immigrants, שיטות Scaling אחרות ועוד – כל גיוון אפשרי מאופשר, וגם ניתן לבחירה בתוך ממשק המשתמש.

ייצוג הבעייה: Genetic Problem & Representation

על מנת לממש תמיכה בבעיה חדשה ופתרונה ע"י המנוע הגנטי, דרוש רק מימוש של הממשק הבא וייצוג פרמטרי שלה (הסבר בהמשך)

```
public interface Genetic<A> {
    double fitness(A gene);
    A mate(A x, A y);

    // Allowed to be in place (done on fresh gene generated by mate)
    A mutate(A a);

    Metric<A> metric();
    A randomElement(Random rand);
    String show(A gene);
}
```

וזה כל מה שצריך לממש על מנת להפעיל את המנוע הגנטי על בעייה חדשה.

באופן אוטומטי, ניתן להריץ על הבעיה את כל הטכניקות של האלגוריתם הגנטי, לבחור אילו שיטות גנטיות להפעיל, כגון Local Optima Detection, כל פונקציות הFitness Mapping, להחליף את שיטת הבחירה, המוטציה וההישרדות, ואף ניתן להפעיל את האלגוריתם הMeta Genetic, על המנוע ועל הפרמטרים הספציפיים לבעיה.

Genetic Alg:

GeneticAlg הוא צירוף של הגדרת הבעייה (Genetic) עם מנוע גנטי (GeneticEngine), שניתן להריץ בעזרתו את האלגוריתם הגנטי.

```
class GeneticAlg[A](val genetic: Genetic[A],
                    geneticEngine: GeneticEngine,
                    rand: Random)
```

ייצוגים פרמטריים

רוב המרכיבים של אלגוריתמים גנטיים והמנוע הגנטי מתבססות על פרמטרים,

כגון Elitism Rate, Mutation Size, Top Selection Ratio ועוד.

על מנת לתמוך בהרחבה של הGenetic Framework שפיתחנו, כך שכל הרכיבים המתבססים על פרמטרים של רכיבים אחרים, כגון האלגוריתם הMeta Genetic, ממשק המשתמש המאפשר לשנות פרמטרים, ורכיב האנליזה על הפרמטרים, יוכלו להיות מותאמים אוטומטית להוספות ושינויי פרמטרים ללא שינוי בקוד שלהם,

ובנוסף על מנת לאפשר מודל תכנותי פשוט ועוצמתי, פיתחנו ייצוגים פרמטריים:

```
case class Parametric[+A]
```

מייצג ייצוג פרמטרי של טיפוס A, ייצוג פרמטי המחזיר איבר מטיפוס A.

ייצוגים פרמטריים Parametric[+A] מהווים מבנה אלגברי הנקרא Applicative.

התאמה למבנה אלגברי זה מאפשר לנו לכתוב קוד באופן הבא:

```
def geneticEngin: Parametric[GeneticEngine] =
  for {
    popSize <- intParam("Population Size", default = 100,
                                           minValue = 3, maxValue = 256)
    localOptimaSignal <- localOptimaSignal
    normalGeneration <- normalGeneration
    localOptimumGeneration <- localOptimaGeneration
  } yield new GeneticEngine(localOptimaSignal,
                            normalGeneration,
                            localOptimumGeneration,
                            popSize)

override def genetic: Parametric[Genetic[Params]] =
 for {
  rounds <- intParam   ("Rounds (for stability)", default = 10,
                                                  minValue = 1, maxValue = 50)
  timeLimit <- doubleParam("Time Limit per config (Seconds)" , 0.3)
  pressure <- doubleParam("Pressure: Window of time from min. time", 0.4)
  relief <- doubleParam("Relief: Percent of relief when bumping the time
limit", 0.05)
  ...
} yield new GeneticParams(rounds, pressure, relief, ...)
```

באופן זה, יש לנו גישה בטוחה אל הפרמטרים שאנחנו דורשים (אין צורך בגישות דינאמיות, לדוגמא אם הכל היה מועבר כמערך וניגשים לאינדקסים ספציפיים לכל פרמטר)

ובנוסף, אופן בניית הרכיבים הגנטיים בשיטה שלנו הוא הוא Compositional:

לדוגמא, בבניית הGenetic Engine, התבססנו על מרכיבים שהם פרמטריים מורכבים בעצמם, והמערכת יודעת להתמודד עם איחוד הפרמטרים בעצמה (בזכות מימוש הApplicative).

Applicatives and Monads

על מנת שמבנה כלשהו ייחשב לApplicative, יש צורך ביכולת המימוש של הפונקציות הבאות:

```
trait Applicative[F[_]] {
  def point[A](value: A): F[A]
  def map2[A, B, C](fa: F[A], fb: F[B], f: (A, B) => C): F[C]
}
```

במקרה שלנו, `F = Parametric` הוא Applicative, מכיוון שניתן ליצור ייצוג פרמטרי לכל ערך, וזאת ע"י כך שהוא לא ידרוש שום פרמטרים,

בנוסף יש צורך ביכולת לשלב שתי ייצוגים פרמטריים לייצוג פרמטרי שלישי, וזאת ע"י איחוד רשימת שמות הפרמטרים ופיצול הפרמטרים האמיתיים שיתקבלו בעתיד לשתי הייצוגים הפרמטריים fa וfb הדורשים פרמטרים בעצמם, ושילוב שתי התוצאות שמתקבלות מהם.

בנוסף, כל Applicative חייב לקיים כמה חוקים, שלא נזכיר כאן.

ראה את:

C. McBride and R. Paterson. Applicative programming with effects.

Journal of Functional Programming, 2008.

עבור המאמר המקורי שהציג את המבנה של Applicative.

הדרישה היחידה מApplicative מבחינה סינטקטית היא שכל הרכיבים יהיו בלתי תלויים, כלומר, כל רכיב לא יכול להתבסס על הפרמטרים שבאו לפניו, לדוגמא גודל הTournament לא יכול להתבסס על גודל האוכלוסייה, ושיטת הבחירה עצמה לא יכולה להתבסס על הMutation Rate.

אם היינו רוצים לקבל גם אפשרות הרכבה זו, היינו צריכים שהייצוג הפרמטרי יהיה גם מבנה אלגברי בשם Monad (ראה את Philip Wadler, 1998), אך לצערינו ייצוגים פרמטריים לא מקיימים את דרישות מבנה אלגברי זה.

דוגמא ליתרונות ולנוחות בשימוש בייצוגים פרמטריים בבחירת מנוע גנטי:

```
println("# Choose The Parent Selection Algorithm:")
println(
  """1. Top Selection (*)
    |2. Roulette Wheel Selection - RWS
    |3. Stochastic Universal Sampling - SUS
    |4. Ranking
    |5. Tournament
  """)
readInt("Choose a parent selection strategy (default 1): ", 1) match {
  case 1 => Instances.topSelection
  case 2 => Instances.rws
  case 3 => Instances.sus
  case 4 => Instances.ranking
  case 5 => Instances.tournament
  case _ => chooseParentSelection()
}
```

ממשק המשתמש

```
Genetic Algorithms Lab 2 by Ilan Godik & Yuval Alfassi

1. Genetic Algorithm
2. Hill Climbing - String matching
3. Baldwin's Effect
Please choose what you want to run: 1

1. String searching
2. Function optimization
3. N-Queens
4. Knapsack
What problem do you want to solve? 3
```

Problem specific settings:

```
Enter the 1st item's weight (0 to stop): 0.22
Enter the 1st item's value (0 to stop): 0.53
Enter the 2nd item's weight (0 to stop): 0.33
Enter the 2nd item's value (0 to stop): 0.77
Enter the 3rd item's weight (0 to stop): 0.1
Enter the 3rd item's value (0 to stop): 0.1
Enter the 4th item's weight (0 to stop): 1.2
Enter the 4th item's value (0 to stop): 3
Enter the 5th item's weight (0 to stop): 1.5
Enter the 5th item's value (0 to stop): 4
Enter the 6th item's weight (0 to stop): 0
Enter the maximum weight: 10
Enter the solution if you know it (nothing if not): 26.36
```

Genetic Framework Menu:

```
1. run     - Run Knapsack
2. params  - Change Parameters of the Genetic Algorithm
3. engine  - Choose the Genetic Engine Algorithms
4. opt     - Optimize Parameters of the Genetic Algorithm
5. analyse - Create a statistical report of the Genetic Algorithm
6. bench   - Benchmark the Genetic Algorithm
7. main    - Return to the main menu

Enter your selection: run
Enter the maximum runtime in seconds (default 1.0):
Print best every how many iterations? (default 1, 0 for never)
```

```
Choose board size (default 10): 20

1. PMX - Partially Matched Crossover (*)
2. OX  - Ordered Crossover
3. CX  - Cycle   Crossover

Choose a mating algorithm (default 1): 2

1. Displacement
2. Exchange (*)
3. Insertion
4. Simple Inversion
5. Complex Inversion
6. Scramble

Choose a mutation algorithm (default 2): 2
```

Can customize Optimization Engine:

```
1. run     - Run Knapsack *Optimization*
2. params  - Change Parameters of the Genetic Algorithm
3. engine  - Choose the Genetic Engine Algorithms
4. opt     - Optimize Parameters of the Genetic Algorithm
5. analyse - Create a statistical report of the Genetic Algorithm
6. bench   - Benchmark the Genetic Algorithm
7. main    - Return to the main menu
```

Example run (Meta Genetic):

```
Best (353): Params: (ints: (Population Size ->  13), doubles: (Elitism Rate -> 0.914, Gene Similarity Threshold -> 0.207, Local Optimum: Elitism Rate -> 0.143, Local Optimum: Hyper Mu
Best (354): Params: (ints: (Population Size ->  10), doubles: (Elitism Rate -> 0.861, Gene Similarity Threshold -> 0.196, Local Optimum: Elitism Rate -> 0.135, Local Optimum: Hyper Mu
Best (355): Params: (ints: (Population Size ->   6), doubles: (Elitism Rate -> 0.894, Gene Similarity Threshold -> 0.271, Local Optimum: Elitism Rate -> 0.133, Local Optimum: Hyper Mu
Best (356): Params: (ints: (Population Size ->  52), doubles: (Elitism Rate -> 0.852, Gene Similarity Threshold -> 0.270, Local Optimum: Elitism Rate -> 0.082, Local Optimum: Hyper Mu
Best (357): Params: (ints: (Population Size ->   3), doubles: (Elitism Rate -> 0.885, Gene Similarity Threshold -> 0.205, Local Optimum: Elitism Rate -> 0.218, Local Optimum: Hyper Mu
Best (358): Params: (ints: (Population Size ->  27), doubles: (Elitism Rate -> 0.895, Gene Similarity Threshold -> 0.200, Local Optimum: Elitism Rate -> 0.000, Local Optimum: Hyper Mu
Best (359): Params: (ints: (Population Size ->  18), doubles: (Elitism Rate -> 0.919, Gene Similarity Threshold -> 0.211, Local Optimum: Elitism Rate -> 0.091, Local Optimum: Hyper Mu
Best (360): Params: (ints: (Population Size ->  20), doubles: (Elitism Rate -> 0.875, Gene Similarity Threshold -> 0.171, Local Optimum: Elitism Rate -> 0.148, Local Optimum: Hyper Mu
Best (361): Params: (ints: (Population Size ->   4), doubles: (Elitism Rate -> 0.911, Gene Similarity Threshold -> 0.221, Local Optimum: Elitism Rate -> 0.071, Local Optimum: Hyper Mu
Best (362): Params: (ints: (Population Size ->  26), doubles: (Elitism Rate -> 0.938, Gene Similarity Threshold -> 0.225, Local Optimum: Elitism Rate -> 0.091, Local Optimum: Hyper Mu
Best 5:
Params: (ints: (Population Size ->  26), doubles: (Elitism Rate -> 0.938, Gene Similarity Threshold -> 0.225, Local Optimum: Elitism Rate -> 0.091, Local Optimum: Hyper Mutation Rate
Params: (ints: (Population Size ->  24), doubles: (Elitism Rate -> 0.874, Gene Similarity Threshold -> 0.220, Local Optimum: Elitism Rate -> 0.000, Local Optimum: Hyper Mutation Rate
Params: (ints: (Population Size ->  29), doubles: (Elitism Rate -> 0.903, Gene Similarity Threshold -> 0.211, Local Optimum: Elitism Rate -> 0.040, Local Optimum: Hyper Mutation Rate
Params: (ints: (Population Size ->  36), doubles: (Elitism Rate -> 0.976, Gene Similarity Threshold -> 0.208, Local Optimum: Elitism Rate -> 0.068, Local Optimum: Hyper Mutation Rate
Params: (ints: (Population Size ->   8), doubles: (Elitism Rate -> 0.896, Gene Similarity Threshold -> 0.241, Local Optimum: Elitism Rate -> 0.001, Local Optimum: Hyper Mutation Rate
100088ms, 362 iterations              seed: -689128128356759870
```

Parameter Customization:

```
Enter your selection: 2
1. Population Size = 7
2. Elitism Rate = 0.1685
3. Gene Similarity Threshold = 0.5
4. Local Optimum: Elitism Rate = 0.0
5. Local Optimum: Hyper Mutation Rate = 1.0
6. Local Optimum: Immigrants Rate = 0.5
7. Local Optimum: Top Ratio = 0.8
8. Mutation Probability (For each item when mutating) = 0.5382
9. Mutation Rate = 0.8595
10. Top Ratio = 0.8
Which parameter to change? (0 to skip)  1
Set Population Size = 100
```

Engine Customization:

```
########## Choosing Genetic Engine ##########


### Choosing Normal Generation ###

# Choose The Parent Selection Algorithm:
1. Top Selection (*)
2. Roulette Wheel Selection - RWS
3. Stochastic Universal Sampling - SUS
4. Ranking
5. Tournament

Choose a parent selection strategy (default 1): 2

# Choose Survivor Selection Algorithm:
1. Elitism (*)
2. Elitism with Random Immigrants
Choose a survivor selection strategy (default 1):

# Choose Fitness Mappings:
1. Windowing (*)
2. Exponential Scaling
3. Sigma Scaling
4. Aging (*)
5. Niching
Enter what you want to choose (multiple selection, blank to continue):

# Choose Local Optima Signal:
1. Ignore Local Optima
2. Gene Similarity Detection (*)
3. Fitness Similarity Detection (std. dev)

Choose a local optima signal (default 2): 2

### Choosing Local Optimum Generation ###

# Choose The Parent Selection Algorithm:
1. Top Selection (*)
2. Roulette Wheel Selection - RWS
3. Stochastic Universal Sampling - SUS
4. Ranking
5. Tournament

Choose a parent selection strategy (default 1): 3
```

Analysis:

```
Enter your selection: 5
Enter analysis name:
Queens-20-PMX-Exchange

run    - Run the analysis
params - Change analysis parameters

Enter your selection: params
1. Ints Step Size = 10
2. Rounds = 10
3. Doubles Step = 0.01
4. Max Time per run (seconds) = 0.2
Which parameter to change? (0 to skip)  2
Set Rounds = 30
run    - Run the analysis
params - Change analysis parameters

Enter your selection: run
Outputting analysis to C:\Users\Ilan\Programming\University\AILab
 \analysis\Queens-20-PMX-Exchange
Int param 'Population Size' = 133, Time = 1.878 ms.
Int param 'Population Size' = 63, Time = 2.407 ms.
Int param 'Population Size' = 193, Time = 2.463 ms.
Int param 'Population Size' = 73, Time = 1.219 ms.
Int param 'Population Size' = 143, Time = 1.926 ms.
Int param 'Population Size' = 203, Time = 2.680 ms.
Int param 'Population Size' = 83, Time = 1.866 ms.
```

Benchmarking:

```
1. run     - Run Knapsack
2. params  - Change Parameters of the Genetic Algorithm
3. engine  - Choose the Genetic Engine Algorithms
4. opt     - Optimize Parameters of the Genetic Algorithm
5. analyse - Create a statistical report of the Genetic Algorithm
6. bench   - Benchmark the Genetic Algorithm
7. main    - Return to the main menu

Enter your selection: bench
Enter the number of rounds (1000 default): 20000
Enter the time limit per run (0.3 default):
0.5539 ms
```

שאלה 1: מדידת זמנים וחקר ביצועים

בהרצת האלגוריתם הגנטי, מודפס זמן הריצה לכל Generation ובנוסף הזמן הכולל, במילי-שניות, כנדרש בתרגיל.

Java לא מספק גישה לclock counter של המעבד, לכן חייבים לעטוף את פקודת המכונה שמשיגה מידע זה ע"י הJNI – Java Native Interface, אך גישה מסוג זה לרוב תיקח זמן רב מדי, לעיתים אפילו זמן ארוך יותר מאשר זמן הריצה הכולל של האלגוריתם הגנטי, (למשל במציאת מינימום לפונקציה אנחנו מתכנסים לפתרון בחצי מילי שנייה), ולכן לא כללנו את מידע זה.

בנוסף, רוב זמן האמת שמוצג למשתמש הוא לא זמן מייצג, מכיוון שההדפסות עצמן למסך לוקחות את הרוב המוחלט של הזמן. לכן הצגנו אפשרות לעשות benchmarking לזמן הריצה הכולל של האלגוריתם הגנטי כל פלט חיצוני, בעזרת כמות גדולה של הרצות חוזרות (לדוגמא 20,000 הרצות של האלגוריתם הגנטי בשלמותו), ובביצוע במקביל על כל ליבות המעבד, וכל זאת כדי לקבל זמן ריצה המייצג היטב את זמן ההתכנסות של האלגוריתם הגנטי, וגם מבלי לחכות זמן ארוך מדי לתוצאות הbenchmarking.

מעבר להדפסת הזמן לריצה על פרמטרים ספציפיים, המנוע הגנטי מכיל כלי אנליזה של פרמטרים הפועל באופן הבא:

- לאחר בחירת בעיה ומנוע גנטי, נרצה לנתח כיצד משפיע כל פרמטר של הבעיה על זמן ההתכנסות ויציבות ההתכנסות.
- לשם כך, תחילה נבחר פרמטרים התחלתיים טובים, ידנית או על ידי האלגוריתם הMeta Genetic שימצא שיערוך לקבוצת פרמטרים טובה.
- לשם האנליזה, המשתמש בוחר את הדיוק הרצוי לפרמטרים ממשיים ושלמים (גודל הצעד בין כל זוג דגימות לפרמטר)
- בנוסף, המשתמש בוחר את מידת הביטחון שהוא רוצה בתוצאות, ע"י בחירת מספר הRounds שזה יריץ את האלגוריתם הגנטי תחת אותם הפרמטרים. בחירת בטחון טוב, לדוגמא 30 ריצות לכל קונפיגורציה היא חשובה מאוד על מנת לקבל תוצאות משמעותיות.
- האנליזה עוברת על כל פרמטר בנפרד, ומשנה אותו לכל הערכים במרחק צעד אחד מהשני – מתבצע שינוי של פרמטר יחיד כל פעם מהקונפיגורציה שקורבה לאופטימלית, וכך אנו בודקים את השפעת משתנה זה על ביצועי המנוע הגנטי.
- כל התוצאות נפלטות לקבצי csv לשם המשך ניתוח ידני והפקת גרפים, בעזרת כלי נוסף שבנינו.

שיפור ביצועי הGenetic Engine

Parallelism:

השתמשנו בתכנות מקבילי רבות כדי לשפר את ביצועי האלגוריתם הגנטי ומרכיבים אחרים בGenetic Framework: מיון גנים מקבילי, אנליזה הרצה על פרמטרים שונים במקביל, Benchmarking המריץ את אותו האלגוריתם הגנטי פעמים רבות במקביל,
והאלגוריתם הMeta Genetic, המריץ את כל הRounds שלו במקביל. (המשמשים לקבל ביטחון ביציבות קבוצת הפרמטרים)

Profiling:

במשך כל הפיתוח עקבנו באופן צמוד בעזרת Profiling עם הכלי JVisualVM על רגרסיות ביצועים וביצועי האלגוריתם הגנטי על מנת למצוא צוואר בקבוק שניתן לשפר.

שיפור ביצועים באלגוריתמי AI מהווים חלק מנכונות האלגוריתם: ככל שהמערכת תפעל מהר יותר, כך הסיכויים לקבל פתרון טוב, נכון או אופטימלי בזמן ריצה סביר ואנושי גדל.

במקרים רבים ניתחנו את תוצאות הקומפילציה על מנת להבין היכן מתבזבז וכיצד לשפר זאת,

ובנוסף עשינו אופטימיזציות ידניות כגון class inlning, והמנעות מהקצאות ע"י שימוש חוזר בזכרון קבוע.

אלגוריתמים יעילים:

חיפושנו אלגוריתמים יעילים יותר לחלק מהרכיבים הגנטיים, כמו לדוגמא:

אלגוריתם streaming לחישוב Standard Deviation (שרץ המון, בזיהוי אופטימום לוקאלי),
ואלגוריתם RWS הרץ בזמן $O(1)$ (ראה את הפרק על שיטות בחירה).

שאלה 2: שיטות בחירה

RWS: אלגוריתם הRWS דוגם את הגנים בהסתברות $\frac{f}{\sum f}$ – יחס ישיר לfitness.

מימשנו את אלגוריתם הבחירה RWS בעזרת Stochastic Acceptance שרץ בזמן $O(1)$:

Roulette-wheel selection via stochastic acceptance [Lipowski, Lipowska, 2011]

http://arxiv.org/abs/1109.3627

קוד:

```
public <A> A chooseSingleParent(Population<A> population, Random rand) {
    int popSize = population.population.length;
    double maxFitness = maxFitness(population);
    int index;
    do {
        index = rand.nextInt(popSize); // 1/N probability to choose anyone
    } while (rand.nextDouble() >=
          (1 - population.population[index].fitness) / maxFitness);
    return population.population[index].gene;
}
```

* נצפו שיפורים טובים במיוחד בביצועים בעזרת RWS בבעיית מציאת המינימום לפונקציה, בשתי הפונקציות.

השוואה של שיטה תחת פרמטרים אופטימיים (ע"פ האלגוריתם הMeta-Genetic), עם בדיקת יציבות ע"י ממוצע של 20,000 הרצות של האלגוריתם הגנטי:
(זמן התכנסות לפתרון האופטימלי)

Top Selection: 0.55 ms

RWS: 0.35 ms

SUS: 0.45 ms

Tournament: 0.45 ms

Ranking: בחירה כמו בRWS, אך אנחנו מתחשבים אך ורק בטיב היחסי של הגן במערך ממוין:

כך אנחנו נותנים הזדמנות גבוהה יותר לגנים לוא דווקא טובים להיבחר,

ובנוסף, אם פונקציית הfitness לא מאוזנת וההבדל בין fitness של גנים גדול מאוד, אז במקום שתהיה העדפה מאוד חזקה לקצת הגנים הטובים, כעת לכולם יש הסתברות יותר הוגנת להיבחרות, ובכך אנחנו מגדילים את הDiversity.

קוד:

```
public <A> A chooseSingleParent(Population<A> population, Random rand) {
    int popSize = population.population.length;
    double maxFitness = popSize - 1;
    int index;
    do {
        index = rand.nextInt(popSize); // 1/N probability to choose anyone
    } while (rand.nextDouble() >= index / maxFitness);
    return population.population[index].gene;
}
```

SUS:

SUS מבצע בחירה של גנים בהתאם לfitness שלהם, ומספר הבחירות פרופורציוני בדיוק לfitness כך שאם יש לנו fitnessים של 0.5,0.25,0.25 אז ייבחרו בדיוק 50% אבות מהגן הראשון, בדיוק 25% מהגן השני ו25% מהגן השלישי.

בעזרת אלגוריתם בחירה זה, אנו מבטיחים הופעה גם של גנים פחות טובים, כל עוד האוכלוסייה שאנו בוחרים גדולה מספיק, ובנוסף ייתכן שגן נדיר אף יותר ייבחר אם אנחנו פגענו בסיבוב ההתחלתי הראשון במיקום, כך שבהזזות של 1/size נגיע אליו.

קוד:

```
public <A> Supplier<A> chooseParents(Population<A> population, int size, Random
rand) {
    // Calculate the sum of all fitness values.
    double fitnessSum = 0;
    for (Gene<A> candidate : population.population) {
        fitnessSum += (1 - candidate.fitness);
    }
    double startPos = rand.nextDouble() * fitnessSum;
    RoulettePosition initialPosition = consume(startPos, new
RoulettePosition(0, 1 - population.population[0].fitness), population);
    double stepSize = fitnessSum / size;
    Supplier<A> supplier = new Supplier<A>() {
        RoulettePosition pos = initialPosition;
        @Override
        public A get() {
            int i = pos.index;
            pos = consume(stepSize, pos, population);
            return population.population[i].gene;
        }
    };
    List<A> parents =
Stream.generate(supplier).limit(size).collect(Collectors.toList());
    Collections.shuffle(parents);
    Iterator<A> iterator = parents.iterator();
    return iterator::next;
}

private static <A> RoulettePosition consume(double amount, RoulettePosition
pos, Population<A> pop) {
    int i = pos.index;
    double remaining = pos.remaining;
    while (remaining <= amount) {
        int nextIndex = (i + 1) % pop.population.length;
        amount = amount - remaining;
        i = nextIndex;
        remaining = 1 - pop.population[nextIndex].fitness;
    }
    return new RoulettePosition(i, remaining - amount);
}
```

שיטות Scaling

מימשנו 5 פונקציות Fitness Mapping, ביניהם כל שיטות הScaling:

- Windowing
- Exponential Scaling
- Sigma Scaling
- Aging Model
- Niching

כל שיטות אלה מומשו בתור Fitness Mappings:

טרנספורמציות על הfitness של כל הגנים באוכלוסיה, בהתבסס גם על שאר האוכלוסיה:

```
public interface FitnessMapping {
    <A> double mapFitness(Metric<A> metric,
                          Population<A> population,
                          Gene<A> gene);
}
```

המנוע הגנטי מחזיק את כל פונקציות המיפוי שאנו רוצים שהמנוע ישתמש בהם, ומפעיל אותם אחד אחד על כל הגנים.

המשתמש יכול לבחור כל תת קבוצה של פונקציות מיפוי לשימוש במנוע הגנטי שלו. (כולל בUI)

```
# Choose Fitness Mappings:
1. Windowing (*)
2. Exponential Scaling (*)
3. Sigma Scaling
4. Aging (*)
5. Niching
Enter what you want to choose (multiple selection, blank to continue): 2
```

שיטות הScaling:

1. Windowing – הפחתת הfitness הגרוע ביותר, אך לא עד ל1 (הכי גרוע) כי אז האפשרות הזו לעולם לא תבחר בRWS, וזאת בכדי ליצור בחירה יותר אחידה והוגנת, ובכך אנו מגדילים את הDiversity.

קוד:

```
public <A> double mapFitness(Metric<A> metric, Population<A> population,
Gene<A> gene) {
    // return 1 - ((1 - gene.fitness) - (1 - population.worstMaxFitness()));
  if (gene.fitness == 0)
    return 0;
  else {
    double fitness = 1 + gene.fitness - population.worstMaxFitness() - epsilon;
    return Math.max(fitness, epsilon);
  }
}
```

2. Exponential Scaling – שורש הfitness, כדי להפחית את השפעתם הקיצונית של הגנים בעלי fitness טוב מאוד.

קוד:

```
public <A> double mapFitness(Metric<A> metric, Population<A> population,
Gene<A> gene) {
    return 1 - Math.sqrt(1 - gene.fitness);
}
```

3. Sigma Scaling – התחשבות במרחק מהתוחלת ביחס לסטיית התקן, כדי לדרוש Diversity – שיהיו לנו גנים שונים ככל האפשר

קוד:

```
public <A> double mapFitness(Metric<A> metric, Population<A> population,
Gene<A> gene) {
    if(gene.fitness == 0.0) return 0;
    if(population.fitnessStdDev() == 0) return 0.99;
    // 1/g such that it's a minimization problem.
    return Math.min(0.99, Math.max(epsilon,
          1 / (Math.abs(gene.fitness - population.fitnessAvg()) / 2 *
                population.fitnessStdDev())));
}
```

לא אפשרנו שלגנים יהיה fitness 1 פה, כדי שעדיין הם יהיו ניתנים לבחירה מתוך RWS.

Aging Model:

לכל גן נשמור את הגיל שלו, כך שהוא נולד בגיל 0, ובכל דור גילו עולה ב1.

Aging משנה את הfitness כך שגנים צעירים מקבלים penalty, ככל שהם מתבגרים עד לגיל הבגרות הfitness שלהם משתפר, ומאז ככל שהם מזדקנים, יחס הfitness קטן חזרה לגיל הצעירים.

הFitness Mapping של Aging מתבסס על שתי פרמטרים:

```
public final int matureAt;
// Influence: multiplied by a number in the range [1-agingInfluence, 1]
public final double agingInfluence;
```

קוד:

```
public <A> double mapFitness(Metric<A> metric, Population<A> population,
Gene<A> gene) {
    int age = gene.age;
    if(matureAt == 0) return age;
    // Scale x such that matureAt goes to 0.5.
    else
        return Math.max(0, Math.min(1,
          f(1 - agingInfluence, (double) age / (2 * matureAt))));
}

// A function on [0, 1] that intersects (0,h), (0.5, 1), (1,h)
private static double f(double h, double x) {
    double a = 4 * h - 4;
    double b = -a;
    double c = h;
    return a * x * x + b * x + c;
}

def aging: Parametric[Aging] =
  for {
   matureAge <- intParam("Maturity Age",default = 5,minValue = 0,maxValue = 20)
   agingInfluence <- doubleParam("Aging influence", 0.3)
  } yield new Aging(matureAge, agingInfluence)
```

האלגוריתם הMeta Genetic

אלגוריתם הMeta Genetic שלנו, ובשמו הנוסף, Genetic Params, הוא אלגוריתם גנטי השואף לעשות מינימיזציה של זמן הריצה עד להתכנסות של אלגוריתם גנטי אחר, על ידי חיפוש פרמטרים אופטימיים עבורו.

בזכות הייצוגים הפרמטריים, האלגוריתם המטא-גנטי יכול לקבל Parametric[GeneticAlg[_]] ולעבוד על אלגוריתם גנטי (הכולל גם את הבעייה והמנוע הגנטי גם לדורות רגילים וגם במינימום לוקאלי), וגם יש לו גישה לקבוצה לא קבועה של פרמטרים של הבעייה והמנוע הגנטי.

האלגוריתם הMeta-Genetic הוא הבעייה הגנטית המורכבת ביותר מבחינת כמות המרכיבים והיכולות שלו. להלן הפרמטרים שלו:

```
rounds              <- intParam   ("Rounds (for stability)", default = 10,
                                          minValue = 1, maxValue = 50)
intsMutationSize    <- doubleParam("Ints Mutation Size"              , 0.1)
doublesMutationSize <- doubleParam("Doubles Mutation Size "          , 0.1)
mutationRate        <- doubleParam("Mutation Rate"                   , 0.5)
timeLimit           <- doubleParam("Time Limit per config (Seconds)" , 0.3)
pressure            <- doubleParam("Pressure: Dynamic window of time from
                                    min. time, [0.1,1] multiplied by 10", 0.4)
relief              <- doubleParam("Relief: Percent of relief when bumping
                                   the time limit", 0.05)
```

<u>Rounds</u>: עבור כל קונפיגורציה של פרמטרים, האלגוריתם הגנטי מלמטה מורץ כמה פעמים כדי לוודא זמן טוב ולקבל פתרונות יציבים, לא מזל חד פעמי בזכות אקראיות.

<u>Ints/Doubles Mutation Size, Mutation Rate</u>: נסמן בβ. כל מוטציה נעשית ע"י בחירת מספר בין $[min, \beta \cdot max]$ עבור השלמים ו$[0, \beta \cdot 1]$ עבור הממשיים, וגודל זה מוסף או מופחת מהפרמטר המתאים: מבוצע בינומית עם הסתברות p = Mutation Rate .

<u>Time Limit</u>: מגבלת הזמן ההתחלתית להרצה יחידה של האלגוריתם הגנטי מתחת.

<u>Pressure</u>: נסמן בρ. לאחר כל הרצה של האלגוריתם הגנטי מתחת, נקטין את חלון הזמן המקסימלי ל $\rho \cdot t_{min}$ - זמן הריצה המינימלי שנתקלנו בו עד כה, וזאת על מנת לכפות Selection Pressure חזק, על מנת שהאלגוריתם הMeta Genetic יתאים את עצמו לבעייה באופן דינאמי, ומכיוון שאיכות הפתרון תלויה בזמן הריצה, גם של גנים אחרים, הגבלה של זמן הריצה חשובה מאוד.

<u>Relief</u>: אם חלון הזמן קטן יותר מדי מהPressure, קונפיגורציות רבות יגיעו למגבלת הזמן שלהם. לכל גן שהגיע למגבלת הזמן שלו, חלון הזמן שלו גדל פי $1 + relief$.

<u>Fitness</u>: הFitness של קונפיגורצית פרמטרים מתפלג לינארית בין $[0,0.5]$ אם הוא סיים לפי זמן הריצה שלו בחלון הזמן, ובין $[0.5,1]$ אם הוא לא סיים, לפי הfitness שהגיע אליו האלגוריתם הגנטי מלמטה. במקרה שיש קונפיגורציה שמגיעה לחלון הזמן, נפסיק להריץ עליה Rounds נוספים כדי לא לעכב את שאר הגנים.

* וכל אלה בנוסף לפרמטרים הרגילים של המנוע הגנטי שמריץ את הMeta Genetic Algorithm, שגם מתמודדים עם מינימום לוקאליים באלגוריתם זה וכו'.

<u>שאלה 3:</u> מטריקת מרחק והאלגוריתם הMeta Genetic

מרחק בין גנים ממומש לכל ייצוג בעייה שיש לנו, ע"י הממשק הבא:

```
public interface Metric<A> {
    double distance(A x, A y);
}
```

אינווריאנט – מרחק בין גנים יהיה מנורמל לערכים שבין 0 ל-1 – דבר שיקל על האלגוריתמים שמשתמשים במרחק.

המרחק בין גנים הכרחי לשם שיפור הDiversity באופן יותר חכם מאשר רק עם fitness – וזאת בעזרת גיוון אמיתי בין הגנים עצמם.

בכך שאנו נדרוש מרחקים גדולים בין גנים, אך כולם יכולים להיות בעלי fitness מאוד טוב, אנו מאפשרים גידול Local Minima רבים בו זמנית באותו הGeneration, ובכך הם יכולים גם לתרום לפתרון הבעייה בעזרת שילובים של features מכל מינימום לוקאלי.

בנוסף, כך אנו נקבל יותר פתרונות אופטימליים מאשר אחד, מה שהיה קורה אם לא היינו דורשים גיוון בגנים עצמם, וכל הPopulation היה מתכנס לגן מסוג יחיד.

<u>מאוד רואים תופעה זו של **שלמות ופתרונות רבים**</u> באלגוריתם הMeta Genetic, בחיפוש על פרמטרים הכוללים גם זיהוי והתמודדות עם Local Minimum, במציאת הפרמטר של Gene Similarity Threshold של הבעייה שעליה עושים אופטימיזציה: מהו סף הדימיון שממנו אנו נכנסים למצב התמודדות עם Local Minima.

ברוב הפעמים, האלגוריתם הMeta גנטי ימצא 2 Thresholds אופטימליים לכל בעייה, ועבורם גם התפלגות שונה לשאר הפרמטרים.

בנוסף, שמנו לב שהאלגוריתם הMeta Genetic תמיד התכנס מאוד חזק לפרמטרים הבאים:

Local Optimum: Hyper Mutation: >99% (or a very large value)

Local Optimum: Elitism Rate: <1% (or a very small value)

והדבר מאוד אינטואיטיבי ונכון, מכיוון שאם אנחנו במינימום לוקאלי, אז כדי לצאת ממנו צריך להכניס הרבה אקראיות, ולהקטין את ההיצמדות למינימום הלוקאלי הנוכחי, וזה באמת עובד, כי עבור פרמטרים אלה (ושאר הפרמטרים שנבחרו), אנו מקבלים את זמן ההתכנסות האופטימלי והטוב ביותר.

המטריקות

על מנת לרשום פונקציות מרחק טובות, או אף 'נכונות' אנו הלכנו לפי הגישה הבאה:

מרחק בין שתי גנים אמור לייצג את אורך ה'מסלול' במרחב בו יש לנו פעולות שינוי מתאימות לבעייה, כך שאנחנו עוברים מגן אחד לשני לאורך המסלול.

בעיית חיפוש מחרוזת:

מרחק ההאמינג של המחרוזות מנורמל ביחס לאורך המילה.

קוד:

```
public static double distance(char[] elem, char[] target) {
    int len = Math.min(elem.length, target.length);
    int fitness = 0;
    for (int i = 0; i < len; i++) {
        fitness += invIndicator(elem[i] == target[i]);
    }
    return (double) fitness / target.length;
}
```

בעיית n המלכות:

הסתכלנו על המרחק שבין לוחות כהפרש שבין סכום כל המרחקים שבין מלכות סמוכות על הלוח. מאחר שהייצוג שלנו למלכות על ידי פרמוטציה הינו **אינווריאנטי לשיקוף ואינווריאנטי ל shift אופקי ואנכי**, רצינו מרחק שייצג את הפרמוטציה בהתאם, כך שנתחשב ללוחות כשקולים או דומים תחת אינווריאנטים אלה.
רצינו ייצוג של ה'דרך' או ה'מסלול' שצריך כדי לעבור מלוח ללוח: הזזה של מלכה אחת ביחס לשניה – השכנה שלה, עד לoffset שיש בלוח השני.

קוד:

```
override def distance(x: QueenPermutation, y: QueenPermutation): Double = {
  var deltas = 0
  val length: Int = x.permutation.length
  for (i <- 0 until length) {
    val delta1 = abs(x.permutation(i) - x.permutation((i+1) % length))
    val delta2 = abs(y.permutation(i) - y.permutation((i+1) % length))
    deltas += abs (delta1 - delta2)
  }
  val res = deltas / ((length - 1) * length)
  assert(res >= 0 && res <= 1)
  res
}
```

בעיית השק:

המרחק בין שני מופעים של בעיית השק יהיה המרחק האוקלידי שבין מערכי כמות ה-items שנלקחו, באופן המנורמל ביחס לשק כולו (הסתכלנו על מערך כמויות כנקודה במרחב הn מימדי). ייצוג כזה של מרחק מבטא כמה יש 'לעבור' מלקיחת מוצרים אחת לאחרת. כמה יש לשנות את מערכי הכמויות שלנו מתשובה אחת לשניה.

קוד:

```
override def distance(x: KnapsackElement, y: KnapsackElement): Double = {
  val instance = x.instance
  val capacity = instance.capacity
  def percentFull(index: Int, amounts: Array[Int]): Double = {
    val maxItems: Double = capacity / instance.items(index).weight
    amounts(index) / maxItems
  }
  def normalizedAmounts(amounts: Array[Int]): Array[Double] = {
    Array.tabulate(amounts.length)(i => percentFull(i, amounts))
  }
  val dist = arrayDistanceD(normalizedAmounts(x.amounts),
normalizedAmounts(y.amounts))
  assert(dist >= 0 && dist <= 1)
  dist
}
```

בעיית מציאת מינימום של פונקציה:

המרחק האוקלידי שבין הנקודות החשודות כמינימום.

קוד:

```
override def distance(x: FuncSolution, y: FuncSolution): Double = {
  Distance.euclidianDistance(x.xInRange, x.yInRange, y.xInRange, y.yInRange)
}

def euclidianDistance(x1: Double,y1: Double, x2: Double,y2: Double):Double = {
  sqrt(square(x1 - x2) + square(y1 - y2))
}
```

שאלה 4: זיהוי אופטימום לוקאלי

קריטריון דימיון פרטים לאבחון מינימום לוקאלי:

נרצה להגיד שאנחנו במינימום לוקאלי, אם יש לנו גנים דומים – משמע יש מרחקים קטנים בין הגנים לפי המטריקה הנתונה.

במקום לחשב את כל המרחקים, וזאת בזמן $O(n^2)$, עשינו את השיערוך הבא:

נסתכל על קבוצת המרחקים של כל הגנים מגן מסוים אחד, הנבחר אקראית.

1. אם יש לנו אוכלוסיה מאוד דומה, יהיו לנו גנים רבים שהם במרחק דומה מהגן הספציפי.
2. אם יש לנו אוכלוסיה מגוונת, המרחקים מהגן הספציפי יהיו גם כן מגוונים.

בעייתיות אפשרית היא אם הגן הספציפי נמצא במרכז תת המרחב המטרי של האוכלוסיה, נראה שהאוכלוסיה אחידה מאוד,

ואם הגן הספציפי נמצא בפינת תת המרחב המטרי, כל המרחקים יראו מגוונים מאוד.

נניח שהאקראיות מטפלת לנו בבעייתיות זאת.

פסודו-קוד:

```
class GeneSimilarityDetector(distanceThresh:Double) extends LocalOptimaSignal {
  override def isInLocalOptima[A](metric: Metric[A],
                                  population: Population[A]): Boolean = {
    val statistics = new RunningStat()
    val pivot = population(rand.nextInt(population.population.length))
    population.population.foreach(x =>
        statistics.push(metric.distance(pivot.gene, x.gene)))
    val stdDev = statistics.standardDeviation()
    stdDev < distanceThresh
  }
}
```

במציאות עשינו inlining לכל אלגוריתם ה.Std. Dev בקבלת שיפורי ביצועים ניכרים,

מכיוון שאלגוריתם זה רץ הרבה מאוד.

קריטריון שונות לאבחון מינימום לוקאלי:

כאשר סטיית התקן של ה-fitness של האוכלוסיה קטנה מסף מסויים – אבחנו שאנו נמצאים במינימום לוקאלי.

קוד:

```
case class StdDevLocalOptimaDetector(stdDevThreshold: Double)
                                      extends LocalOptimaSignal {
  override def isInLocalOptima[A](metric: Metric[A],
                                  population: Population[A]): Boolean = {
    val statistics = new RunningStat()
    population.population.foreach(x => statistics.push(x.fitness))
    val stdDev = statistics.standardDeviation()
    return stdDev < stdDevThreshold
  }
}
```

שאלה 5: התמודדות והיחלצות מאופטימום לוקאלי

הגישה שלנו להתמודדות עם אופטימום לוקאלי היא פיצול המנוע הגנטי ל2 מנועים נפרדים:

- האחד לתהליך אבולוציה רגיל
- והשני למקרה שאנו מזהים מינימום לוקאלי

על ידי כך, יהיה אפשר להוסיף שיטות שונות למנוע הגנטי באופטימום לוקאלי, כגון Niching וRandom Immigrants, ולשנות את כל הפרמטרים של המנוע למקרה שאנו באופטימום לוקאלי, לדוגמא Elitism Rate וMutation Rate – כך ממומש Hyper Mutation.

Niching:

Niching הוא פונקציית Fitness Mapping אשר נותנת קנס על גנים דומים – כך נכפה Diversity – גנים חדשים שרחוקים מהגנים הדומים יקבלו ייתרון מבחינת הfitness שלהם.

קוד:

```
public <A> double mapFitness(Metric<A> metric, Population<A> population,
Gene<A> gene) {
    double sumOfSharingFunc = 0;
    for (int i = 0; i < population.population.length; i++) {
        double distance =
                 metric.distance(gene.gene, population.population[i].gene);
        if (distance < sigmaShare)
            sumOfSharingFunc += 1 - Math.pow(distance / sigmaShare, alpha);
    }
    return 1 - ((1 - gene.fitness) / sumOfSharingFunc);
}
```

כאשר alpha וsigmaShare הם פרמטרים בייצוג הפרמטרי של Niching וניתן לשנות אותם מתוך הUI ולעשות עליהם אופטימיזציה.

Random immigrants:

Random Immigrants בצירוף Elitism מהווה שיטת בחירת שורדים אלטרנטיבית לElitism בלבד, וניתן לבחור שיטה זו למנוע של ההתמודדות עם האופטימום הלוקאלי.

כאשר אובחן שאנו נמצאים במינימום מקומי, נוסיף לאוכלוסיה שלנו מספר גנים רנדומיים חדשים וטריים שירעננו את מאגר הגנים שלנו, בתקווה שיהיה בהם מידע ובחירות אלטרנטיביות למה שיש לנו עד כה, ושתכונות טובות אלה ייכנסו לאוכלוסיה וישפרו אותה.

קוד:

ממשק בחירת השורדים לדור הבא:

בהינתן האוכלוסיה הקודמת, וStream אינסופי של ילדים, יש לייצר את האוכלוסיה הבאה.

```
public interface SurvivorSelection {
    <A> void selectSurvivors(Genetic<A> alg,
                             Population<A> population,
                             Population<A> buffer,
                             Function<Integer, Supplier<A>> getChildren,
                             Random rand);
}
```

מיד אחרי האליטיזם יבואו המהגרים הרנדומיים (במידת הצורך)

```
public <A> void selectSurvivors(Genetic<A> alg,
                                Population<A> population,
                                Population<A> buffer,
                                Function<Integer, Supplier<A>> getChildren,
                                Random rand) {
    int popSize = population.population.length;
    int elites = (int) (popSize * elitismRate);
    elitism(population, buffer, elites);

    int immigrants = (int) randomImmigrantsPercent * popSize;
    for (int i = elites; i < (elites + immigrants) && i < popSize; i++) {
        buffer.population[i].gene = alg.randomElement(rand);
        buffer.population[i].age = 0;
    }

    int numChildren = Math.max(popSize - elites - immigrants, 0);
    Supplier<A> children = getChildren.apply(numChildren);
    for (int i = elites; i < popSize; i++) {
        buffer.population[i].gene = children.get();
        buffer.population[i].age = 0;
    }
}
```

פרמטרים לייצוג הפרמטרי:

```
elitismRate          <- doubleParam("Elitism Rate", 0.0)
randomImmigrantsRate <- doubleParam("Immigrants Rate", 0.5)
```

שאלה 6: ניתוחי השיטות הגנטיות

כל הניסויים הבאים נעשו על Queens-15-PMX-Exchange, תחת שינוי שיטה יחידה כל פעם מתוך המנוע הגנטי הבסיסי הבא:

Selection: SUS

Fitness Mappings: None

Survival: Elitism

Local Optima Signal: Gene Similarity

Local Optima Generation:

Survival: Elitism & Random Immigrants

The rest of the engine the same as in the normal generation.

ותחת אופטימיזציה של 100 שניות לכל קונפיגורציה, ועם Benchmark של 20,000 איטרציות, כל הזמנים ומספר הדורות הם עד להתכנסות לפתרון.

Selection Strategies:

Top Selection: 3.38 ms, 20-100 generations (Population Size = 80)

RWS Selection: 4.05 ms, 20-70 generations (Population Size = 150)

SUS Selection: 3.81 ms, 20-100 generations (Population Size = 100)

Ranking: 10.39 ms, 150-500 generations (Population Size = 100)

Tournament: 4.83 ms, 5-30 generations (Population Size = 150, Tournament Size = 3)

הערות:

עבור הבעיה הספציפית הזאת, Top Selection היה הכי מהיר, וTournament היה הכי יציב.

Top Selection הוא טוב גם עבור Diversity וגם עבור Selection Pressure: הוא בוחר מבין אחוז כלשהו מהעליונים, ומביניהם לא נותן עדיפות לפי הfitness, ולכן מאפשר סיכויים שווים גם לגנטים פחות טובים מהאוכלוסיה, ובכך מאפשר סיכוי טוב להכנסת תכונות טובות.

RWS בוחר גנים באופן ממושקל לפי הfitness שלהם, וככל שהfitness טוב יותר, כך סיכוייו להיבחר טובים יותר. אם יש פער גדול בין הfitness של הטובים ביותר לבין הגרועים יותר, הסיכויים של הגרועים להיבחר הוא קלוש. לכן RWS מתמקד יותר בSelection Pressure מאשר Diversity. לשם איזון פערים אלה אנו משתמשים בשיטות Scaling.

SUS בוחר גנים לפי קפיצות בגודל קבוע מסביב לרולטת הבחירה ע"פ הfitness, ונותן סיכוי טוב יותר גם לגנים חלשים להיבחר באופן דטרמיניסטי כאשר מיוצרת אוכלוסיה גדולה. כך שיפרנו במעט את הDiversity מאלגוריתם הRWS.

ניתוחי השיטות הגנטיות 2

Ranking מתעלם לחלוטין מערכי הfitness עצמם ובוחר כמו RWS אך רק לפי המיקום היחסי של הfitnesses, בתקווה לשיפור הDiversity ולחוסר אפליה. במציאות, התעלמות מוחלטת זו מובילה לביצועים נוראיים.

Tournament בוחר תת קבוצה בגודל k וממנה את האיבר הטוב ביותר מביניהם. יש פה שילוב טוב גם של Diversity ע"י כך שאנו בוחרים תת קבוצה אקראית – גם מבין גנים פחות טובים, וגם יש Selection Pressure טוב – כי בוחרים את הגן הטוב ביותר מבין תת הקבוצה. קיבלנו ששיטה זו היא היציבה ביותר ובנוסף היא מסיימת את האלגוריתם במספר הדורות הקטן ביותר.

<u>Fitness Mappings:</u>

Windowing + SUS: 3.11 ms, 30-100 generations (Population Size = 90)

Exponential Scaling: 8.42 ms, 20-100 generations (Population Size = 200)

Sigma Scaling: 60-400/∞ ms, Doesn't always converge, can't optimize.

Aging: 11.90 ms, 30-150 generations (Population Size = 120, Normal Maturity Age = 0, Local Optima Maturity Age = 3, Aging Influence = 0.65)

Niching: ∞ ms.

Windowing מאזן את שיטות הRWS והSUS, ע"י כך שמתחשב רק בחלון הfitnessים האקטואליים בדור זה – החל מהfitness הגרוע ביותר ואילך. בכך אנו מאזנים את הסתברויות הבחירה ומגדילים את הDiversity.

Exponential Scaling מפחית את גודל הייתרון של הגנים הטובים על הפחות טובים ע"י הפעלת פונקציה קעורה, אך עדיין מונוטונית עולה על הfitness – פונקציית שורש, תוך שיפור הDiversity והקטנת הSelection Pressure.

Sigma Scaling לרוב לא מתכנס. שיטה זו שופטת גנים ע"פ הDiversity שלהם בלבד ולא לפי הfitness שלהם: הפרש הfitness מהממוצע חלקי פעמיים סטיית התקן.

Aging נותן penalty לגנים צעירים ולגנים זקנים מדי, בכך נרצה לבסס אוכלוסיה חזקה בגיל מסויים, במטרה גם להגדיל Selection Pressure ע"י בחירת גנים שאנחנו כבר יותר בטוחים בהם, והגדלת הDiversity ע"י השינוי המתמיד של האוכלוסיה הבוגרת. לדעתינו שיטה זו מתאימה לSteady State GA ולא לGenerational GA.

Niching נותן קנס על גנים שנמצאים בנישות גדולות – יש הרבה גנים שדומים להם. בעקרון זה אמור להגביר את הDiversity. הבעייתיות היא שהאלגוריתם לוקח זמן של $O(n^2)$ ורץ **המון** פעמים, ולכן הוא לא פרקטי, לפחות לא באוכלוסיות בגדלים הדרושים לבעיה זו ולבעיות לא טריוויאליות אחרות.

ניתוחי השיטות הגנטיות 3

זיהוי והתמודדות עם מינימום לוקאלי

שיטות אלה עוזרות בכל הבעיות, אך במיוחד בבעיית הMeta Genetic Algorithm:

בעזרתם, אנו מגיעים לפתרונות אופטימליים מרובים בדור האחרון (שהוא הפלט).

בנוסף, שמנו לב לשיפור עצום באיכות הפתרונות שאנחנו מקבלים מהאלגוריתם הMeta גנטי בעזרת שיטות הזיהוי וההתמודדות עם מינימום לוקאלי.

<u>מינימיזציה של פונקציה (פונקצית ניסוי נוספת לבדיקת אופטימיזציה):</u> עם זיהוי והתמודדות עם מינימום לוקאלי: 5.00 ms, ובלי התמודדות עם מינימום לוקאלי: 10.78 ms, עם Hyper Mutation בלבד וללא Random immigrans: 8.71 ms.

לכן השיטות להתמודדות עם מינימום לוקאלי, המכניסות אקראיות ואוכפות הרבה Diversity ברגעים בהם האוכלוסיה מאוד אחידה, עוזרים מאוד ומשפיעים רבות על **זמן ההתכנסות לפתרון האופטימלי**, וכן מאפשרות למצוא פתרונות טובים רבים ובכך עוזרת להשגת **שלמות** עבור האלגוריתם הגנטי.

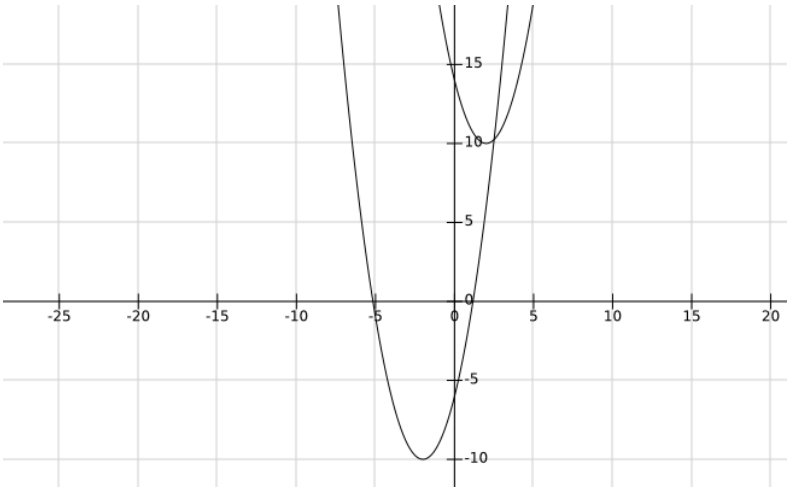
שאלה 7: פרטו אופטימל

פרטו אופטימל עבור בעיית אופטימיזציה מרובת מטרות – Multiobjective Optimization אשר כוללת לרוב מטרות מנוגדות, כוללת את כל הפתרונות הנמצאים על חזית הפרטו אופטימל – הPareto Front, עליו כל הפתרון אופטימלי מהבחינה שלכל פתרון אחר, הוא גרוע מפתרון זה לפחות באחת מפונקציות המטרה.

התבקשנו להציג את הפרטו אופטימל של זוג הפונקציות הבאות:

$$f(x) = (x+2)^2 - 10$$

$$g(x) = (x-2)^2 + 10$$



הפרטו אופטימל עבור שתי פונקציות מטרה אלו הוא בטווח $-2 \le x \le 2$ מכיוון שעבור כל $x \ge 2$ ניתן לשפר את שתי פונקציות המטרה ע"י הפתרון של $x = 2$, ולכל פתרון $x \le -2$ ניתן לשפר את שתי פונקציות המטרה ע"י הפתרון של $x = -2$. עבור כל $-2 \le x \le 2$, בירידת הx משתפרת פונקציית המטרה $f(x)$ ופונקציית המטרה $g(x)$ נהית יותר גרועה.

חשבנו על רעיון לפרוייקט:

איסוף נתוני ביצועים ומחירים מחנויות מחשבים שונות ומיד שנייה על רכיבי מחשב שונים ומחשבים ניידים, ושימוש בחזית הפרטו אופטימל כשירות עזרה לבחירת קניית מחשב

התעניינו גם במנוע הפרטו אופטימל של IBM – Tradeoff Analytics לשם מימוש פרוייקט זה, ונכחנו בהרצאה על מנוע זה ע"י ראש הצוות של הפרויקט מIBM חיפה.

חלק ב – אפקט בולדווין

אפשט בולדווין בא להדגים רעיון אבולוציוני שאומר שהלמידה של יצור במהלך חייב כן משפיעה על הצאצאים, אך באופן עקיף: אלה שלמדו נכון במהלך חייהם ישרדו יותר, ולכן גם בניהם צפויים להיות מותאמים יותר לבעיה, וללמוד אותה היטב, ולשרוד היטב וכך הלאה.

בקונטקסט של אלגוריתמים גנטיים, אפקט בולדווין מדגים רעיון של חיפוש לוקאלי עבור גן, כך שנוכל לצבור ידע נוסף עליו, לקבוע בהתאם את פונקציית המטרה ובכך לכוון את החיפוש למקום הרצוי.

לפי הניסוי, אפקט בולדווין אכן פועל, מכיוון שהאלגוריתם מתכנס לפתרון הנכון.

קוד:

```
public static int localSearchesTimeRemaining(
        byte[] baldwinString, byte[] target, int maxIterations, Random rand) {
    for (int iter = 0; iter < maxIterations; iter++) {
        boolean foundTarget = localSearch(baldwinString, target, rand);
        if(foundTarget) return maxIterations - iter;
    }
    return 0;
}

public static boolean localSearch(
                            byte[] baldwinString, byte[] target, Random rand) {
    for (int i = 0; i < baldwinString.length; i++) {
        byte currentBit = baldwinString[i];
        if(currentBit == QuestionMark) {
            byte setBit = genBit(rand);
            if(target[i] != setBit) return false;
        } else if (currentBit != target[i]) return false;
    }
    return true;
}

override def fitness(gene: Array[Byte]): Double = {
  if(Arrays.equals(gene, target)) 0
  else {
    // do not let the fitness to be 0
    val remainingIterations = Math.min(
         localSearchesTimeRemaining(gene, target, maxIterations, rand),
                                    maxIterations - 1)
    // 1~20
    val rawFitness = 1 + 19.0 * remainingIterations.toDouble / maxIterations
    val normalized = 1 - rawFitness / 20 // 1 -> 1, 20 -> 0
    normalized
  }
}
```

ניתוח הריצה של Baldwin's Effect:

<u>שלב 1:</u>

תחילה יש לנו אוכלוסיה שאין בה גן שצודק בכל הביטים שהוא מתחייב עליהם.

לכן כל הגנים ירוצו עד למכסת הריצה שלהם, ולא יגיעו לתוצאה הנכונה, ויקבלו fitness = 1.

במהלך הMating בשלב זה, יגדל מספר הביטים הלא קבועים באופן מפאת האקראיות, עד שנגיע לגן שבו כל הביטים שעליהם התחייב נכונים.

<u>שלב 2:</u>

כעת, כשיש לנו כבר גן שיכול להגיע לפתרון באופן פוטנציאלי, גן זה יתרבה עם גנים אחרים תוך הגדלת מספר הביטים הנכונים – מכיוון שבקבלת ביט לא נכון, גן זה לא ישרוד הרבה הלאה עם fitness = 1.

בנוסף, החיפוש מכוון לכיוון הגדלת מספר הביטים הנכונים, מכיוון שכך זמן ההגעה לפתרון יהיה נמוך יותר, וכך גם fitness טוב יותר, ויכולת רביה טובה יותר.

בעיה פוטנציאלית בניסוח הניסוי המקורי:

אם לא מכניסים מוטציות לאלגוריתם, לעיתים האלגוריתם יכול להיתקע ולא להתכנס, וזאת מכיוון שבכל האוכלוסיה, אין גנים בהם במיקום מסוים יש את הביט הנכון.

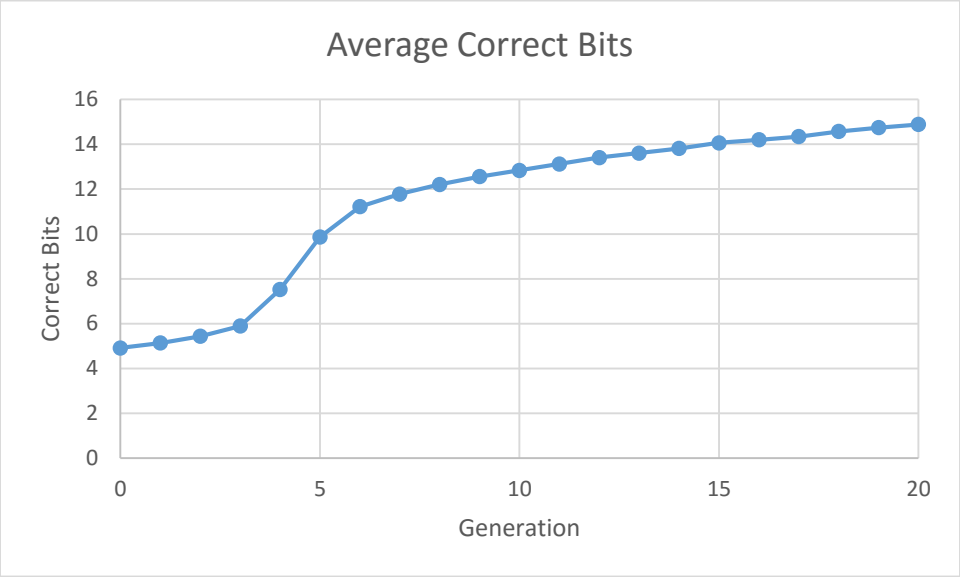
במקרה כזה, לא משנה איזה צירוף של Mating נעשה, לא נוכל להגיע לפתרון הנכון.

לכן יש צורך בMutation כדי שהאלגוריתם ירוץ כהלכה ויתכנס.

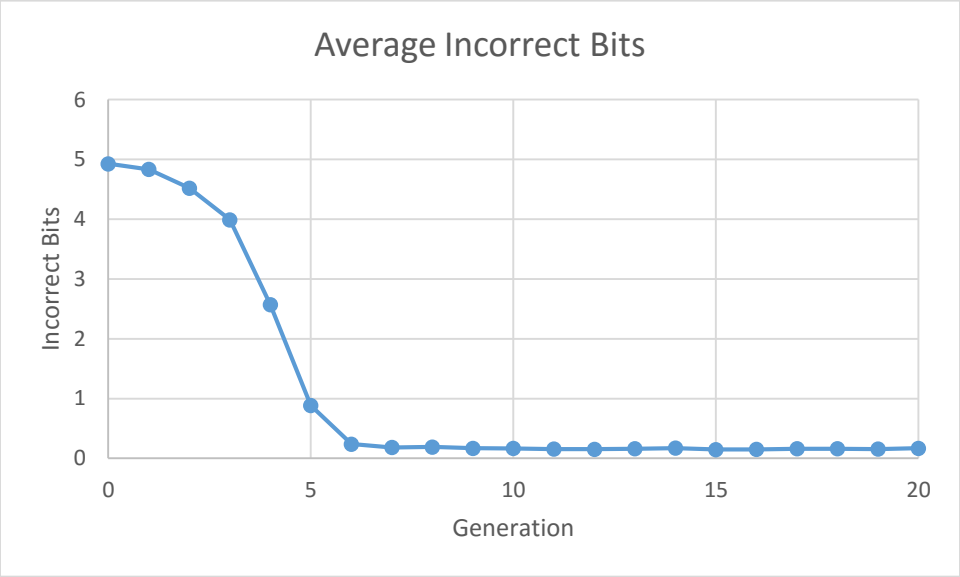
* איפשרנו לשנות את הMutation Rate ולהריץ את אפקט בולדווין תחת כל מנוע גנטי שהו, כמו שאר הבעיות לאלגוריתם הגנטי. ניתן לקבל את סביבת הניסוי המקורי ע"י Mutation Rate = 0 וElitism Rate = 0.

גרפים – Baldwin

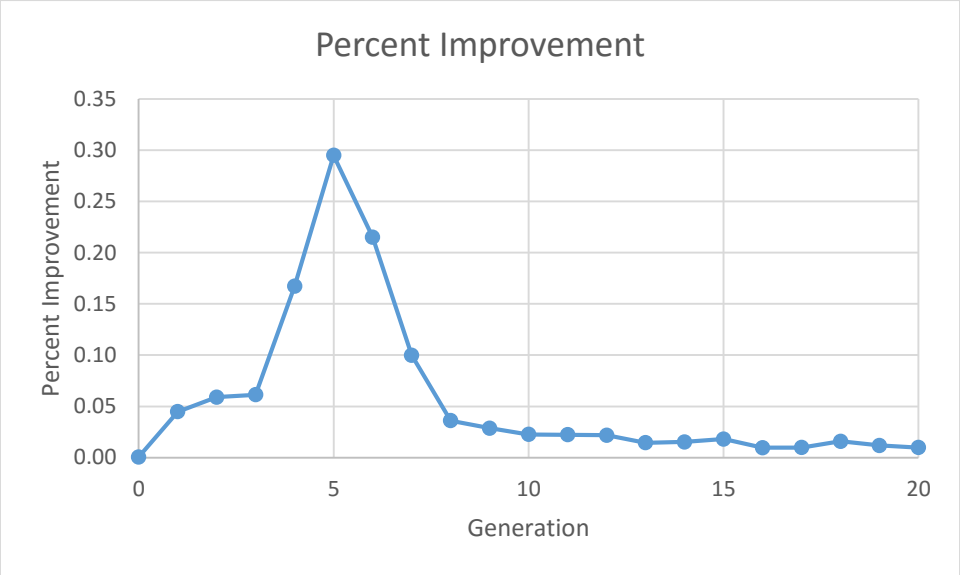
אחוז ביטים נכונים:



אחוז ביטים לא נכונים:



אחוז ביטים נלמדים בין כל זוג דורות:



מסקנה:

אכן נצפה אפקט בולדווין, ישנה למידה דרך ירושה באופן עקיף.

בנוסף, הגרפים מאששים באופן חד את ההשערה שלנו למהלך הריצה של אפקט בולדווין.