מעבדה 2 בבינה מלאכותית

נושא:

אלגוריתמים גנטיים, ממטיים ואבולוציה

­­

מגישים:

1. אילן גודיק, 316315332
2. 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: 50 שעות
* 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:

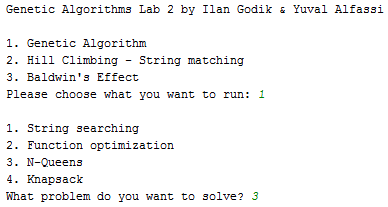
C. McBride and R. Paterson. Applicative programming with effects.

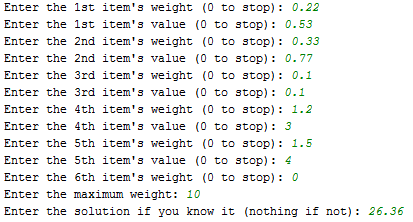
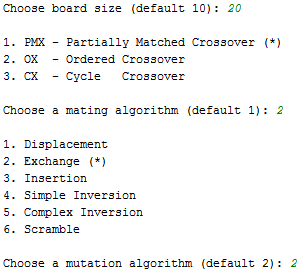
Journal of Functional Programming, 2008.

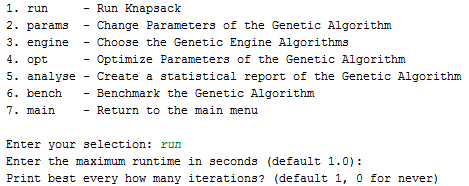
עבור המאמר המקורי שהציג את המבנה של Applicative.

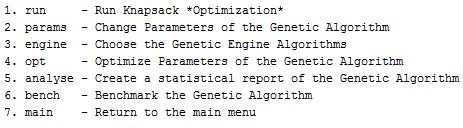
הדרישה היחידה מApplicative מבחינה סינטקטית היא שכל הרכיבים יהיו בלתי תלויים, כלומר, כל רכיב לא יכול להתבסס על הפרמטרים שבאו לפניו, לדוגמא גודל הTournament לא יכול להתבסס על גודל האוכלוסייה, ושיטת הבחירה עצמה לא יכולה להתבסס על הMutation Rate.

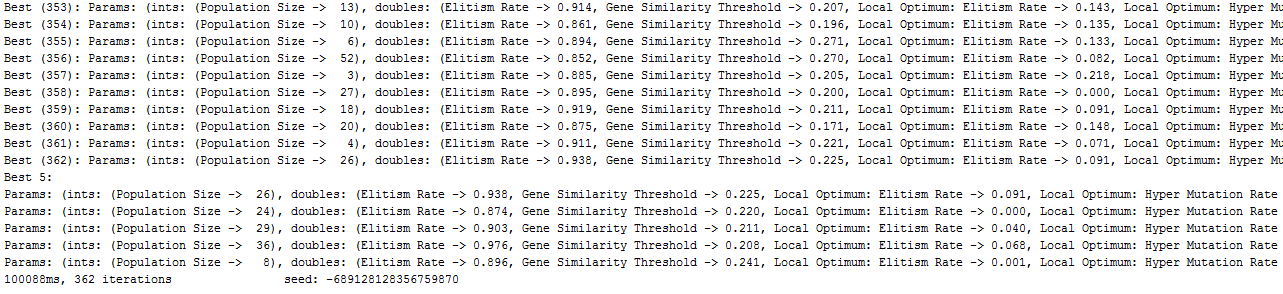
אם היינו רוצים לקבל גם אפשרות הרכבה זו, היינו צריכים שהייצוג הפרמטרי יהיה גם מבנה אלגברי בשם Monad (ראה את Philip Wadler, 1998), אך לצערינו ייצוגים פרמטריים לא מקיימים את דרישות מבנה אלגברי זה.

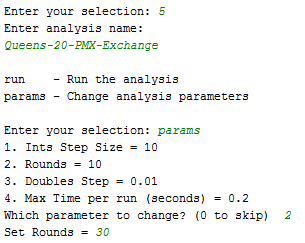
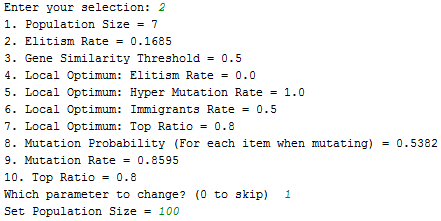
**ממשק המשתמש**

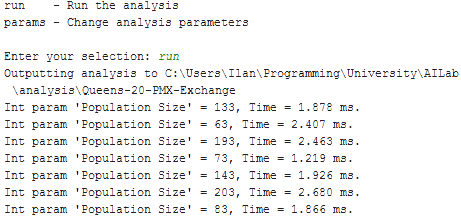
**Problem specific settings:**

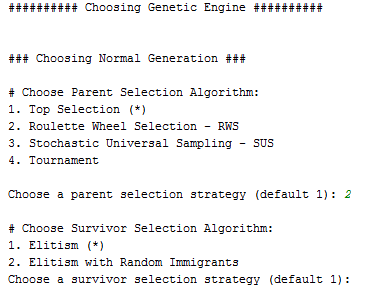
**Genetic Framework Menu:**

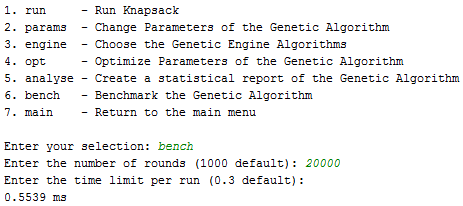
 **Can customize Optimization Engine:**

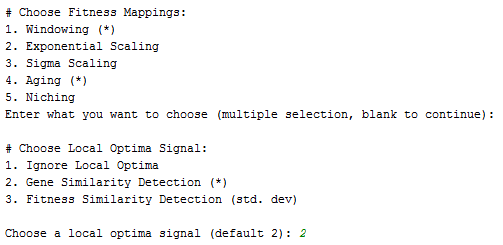
**Example run (Meta Genetic):**

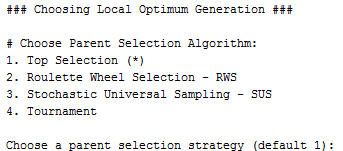
**Parameter Customization: Analysis:**



**Engine Customization:**

** Benchmarking:**

****

****

**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 צווארי 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 (ראה את הפרק על שיטות בחירה).

**שאלה 2: שיטות בחירה**

RWS: אלגוריתם הRWS דוגם את הגנים בהסתברות – יחס ישיר לfitness.

מימשנו את אלגוריתם הבחירה RWS בעזרת Stochastic Acceptance שרץ בזמן :

# Roulette-wheel selection via stochastic acceptance [[Lipowski](http://arxiv.org/find/cs/1/au:+Lipowski_A/0/1/0/all/0/1), [Lipowska](http://arxiv.org/find/cs/1/au:+Lipowska_D/0/1/0/all/0/1), 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 לווא 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:

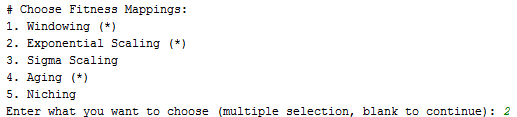
**public interface** FitnessMapping {  
 <A> **double** mapFitness(Metric<A> metric,

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)



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***);  
 }  
}

1. 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**);  
}

1. 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)

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

**Ints/Doubles Mutation Size, Mutation Rate:** נסמן ב. כל מוטציה נעשית ע"י בחירת מספר בין עבור השלמים ו עבור הממשיים, וגודל זה מוסף או מופחת מהפרמטר המתאים: מבוצע בינומית עם הסתברות p = Mutation Rate.

**Time Limit:** מגבלת הזמן ההתחלתית להרצה יחידה של האלגוריתם הגנטי מתחת.

**Pressure:** נסמן ב. לאחר כל הרצה של האלגוריתם הגנטי מתחת, נקטין את חלון הזמן המקסימלי ל - 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.

**Relief:** אם חלון הזמן קטן יותר מדי מהPressure, קונפיגורציות רבות יגיעו למגבלת הזמן שלהם. לכל גן שהגיע למגבלת הזמן שלו, חלון הזמן שלו גדל פי .

**Fitness:** הFitness של קונפיגורצית פרמטרים מתפלג לינארית בין אם הוא סיים לפי זמן הריצה שלו בחלון הזמן, ובין 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'.

**שאלה 3: 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.

מאוד רואים תופעה זו של **שלמות** **ופתרונות רבים** 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 , 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 הStd. 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** ו **sigmaShare** 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:**

**שאלה 7:**

חזית פרטו אופטימל שמצאנו עבור הפונקציות המופיעות להלן הינה

קוד גנטי עבור מציאת הפרטו פרונט:

**חלק ב – אפקט בולדווין**

\*\*\*\*\*\*\*\*\*שאילן יחפור על זה כאן... ליובל אין כוח\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

לפי הניסויים שעשינו אפקט בולדווין כן קורה בלה בלה בלה

קוד:

**override def** fitness(gene: Array[BaldwinBit]): Double = {  
 **val** remainingItarations = BaldwinBitString.*localSearchToBest*(gene, maxIterations, isFinal, rand)  
 **val** rawFitness = 1 + 19.0 \* remainingItarations.toDouble / maxIterations  
 **val** normalized = (20 - rawFitness) / 20  
 *assert*(normalized >= 0 && normalized <= 1)  
 normalized  
}

גרפי הרצה:

לפי מספר הרצות שעשינו, ראינו שהחל משלב מסויים האחוז של ממוצע הביטים הנכונים בכל דור גדל בצורה ניכרת. ההסבר לכך: אפקט בולדווין.

אחוז ממוצע של ביטים נכונים בכל דור:

אחוז ממוצע של ביטים לא נכונים בכל דור:

בהתחלה מתחילים עם 25%, שמתחיל מעט יציב, ואז הולך ויורד עד לכמעט התאפסות. פה ושם יש גנים עם מעט לא נכונים עקב מוטציות וזיווגים

אחוז ממוצע של למידה של ביטים:

כפי שניתן לראות, יותר ביטים נלמדים מאשר לא נלמדים. סך הכל, כמות סימני השאלה הולכת ופוחתת ההסבר לכך: אפקט בולדווין.