מעבדה 1 בבינה מלאכותית

נושא:

מבוא לאלגוריתמים גנטיים

­­

מגישים:

1. אילן גודיק, 316315332
2. יובל אלפסי, 318401015

מנחה:

מר שי בושנסקי



תאריך הגשה:

18 למרץ 2016

**תוכן עיניינים**

דג

**מבוא – המנוע הגנטי**

אלגוריתם גנטי הוא אלגוריתם המדמה את הטבע לשם פתרון בעיית אופטימיזציה ע"י התפתחות אוכלוסיית פתרונות במשך הזמן. האוכלוסייה משתפרת עם הזמן עקב הישרדות והתפתחות הפתרונות הטובים מתוך האוכלוסייה, בתקווה שנגיע לפתרון האופטימלי.

בתרגיל זה פיתחנו Genetic Engine, בעזרתו ניתן לפתח בקלות אלגוריתמים גנטיים שונים לבעיות שונות.

אלגוריתם גנטי צריך להגדיר את ייצוג הפתרונות ולממש 3 פונקציות בסיסיות:

**public interface** Genetic<A> {  
 **double** fitness(A gene);  
 A mate(A x, A y);A mutate(A a);  
}

* **A** מייצג את ייצוג הפתרונות לבעיה אותה אנו מנסים לפתור, לדוגמא מחרוזות או מספרים ממשיים.
* פונקצית הfitness מחזירה עד כמה טוב פתרון מסויים, כך ש1 מסמל שהפתרון הוא הרע ביותר, ו0 מסמל שהפתרון הוא הטוב ביותר. המנוע הגנטי ינסה למצוא פתרון בעל fitness נמוך ככל האפשר.
* פונקציית הmate מכליאה בין שני פתרונות נתונים בקבלת פתרון שלישי, המקבל חלק מן התכונות של כל אחד מהוריו. כך נקווה לקבל את היתרונות של כל אחד מההורים בבן שנוצר. זהו סימלוץ של רבייה בעולם החי.
* פונקציית הmutate צריכה לשנות אקראית את הפתרון הנתון, בתקווה שנקבל פתרון יותר טוב עם מספיק הרצות, ושנצא ממינימום לוקאלי.

בנוסף, אלגוריתם גנטי צריך לבחור את הדברים הבאים:

**class** GeneticAlg[A](alg: Genetic[A], mateStrategy: MateStrategy,

selection: SelectionStrategy, PopulationSize: Int,  
 MaxTimeSecs: Double, rand: Random,  
 randomElement: Random => A, show: A => String)

כאשר MateStrategy מתאר כיצד תיווצר האוכלוסיה הבאה מן האוכלוסיה המקורית, לדוגמא ע"י elitism, רביה בין גנים שונים, ומוטציות. מימשנו MateStrategy בודד, שמקבל את אחוז המוטציות וגודל קבוצת הElitism – מספר הפתרונות הטובים ביותר באוכלוסיה שעוברים לאוכלוסיה הבאה ללא שינוי, בדומה להישרדות בטבע.

SelectionStrategy מתאר אילו הורים לבחור לרבייה בעזרת הממשק הבא:

**public interface** SelectionStrategy {  
 <A> A chooseParent(Population<A> parentsPool, Random rand);  
 <A> **void** populateParentsPool(Population<A> population,

Population<A> parentsPool, Random rand);  
 <A> Population<A> initParentsPool(**int** populationSize);  
}

* populateParentsPool בוחר את אוסף הגנים שישתתפו בבחירת ההורים לmating. (ע"י מילוי הbuffer של parentsPool לשם מניעת הקצאות)
* initParentsPool יוצר אוסף גנים התחלתי, בהתבסס על גודל האוכלוסיה באלגוריתם הגנטי.
* chooseParent בוחר הורה מתוך אוסף ההורים לרוויה.

מימשנו שתי SelectionStrategies:

1. TopSelection(topPercent), אוסף ההורים הוא topPercent האחוזים העליונים הטובים ביותר מתוך האוכלוסיה, והבחירה מתוכם אקראית.
2. Tournament(tournamentSize, p), אוסף ההורים הוא תת קבוצה אקראית בגודל tournamentSize של האוכלוסיה, ומתוכה אנו בוחרים את הגן הכי טוב בהסתברות p, את השני הכי טוב בהסתברות (1-p)p וכך הלאה.

האלגוריתם הגנטי גם דורש את גודל האוכלוסיה וחסם לזמן הריצה של האלגוריתם הגנטי: האלגוריתם הגנטי ירוץ עד שיגיע לfitness של 0 או עד שייגמר הזמן.

אלגוריתמים גנטיים מתבססים על פרמטרים רבים לשם הריצה שלו, לדוגמא גודל האוכלוסיה, אחוז הElitism, אחוז האוכלוסיה העליונה שתתרבה, אחוז המוטציות וגודל המוטציות.

פרמטרים אלה משפיעים רבות על זמן ההתכנסות של האלגוריתם הגנטי לפתרון האופטימלי, ולכן הוספנו אפשרות לעשות אופטימיזציה לפרמטרים של אלגוריתם גנטי עם אלגוריתם גנטי, ולשם כך, כל אלגוריתם גנטי נדרש להצהיר מהם הפרמטים שלו:

**public abstract class** GeneticMain<A> {

**public abstract** Params defaultParams();  
 **public abstract int** intsMax();  
 **public abstract** GeneticAlg<A> alg(Params params, **double** maxTime);

}

­­­

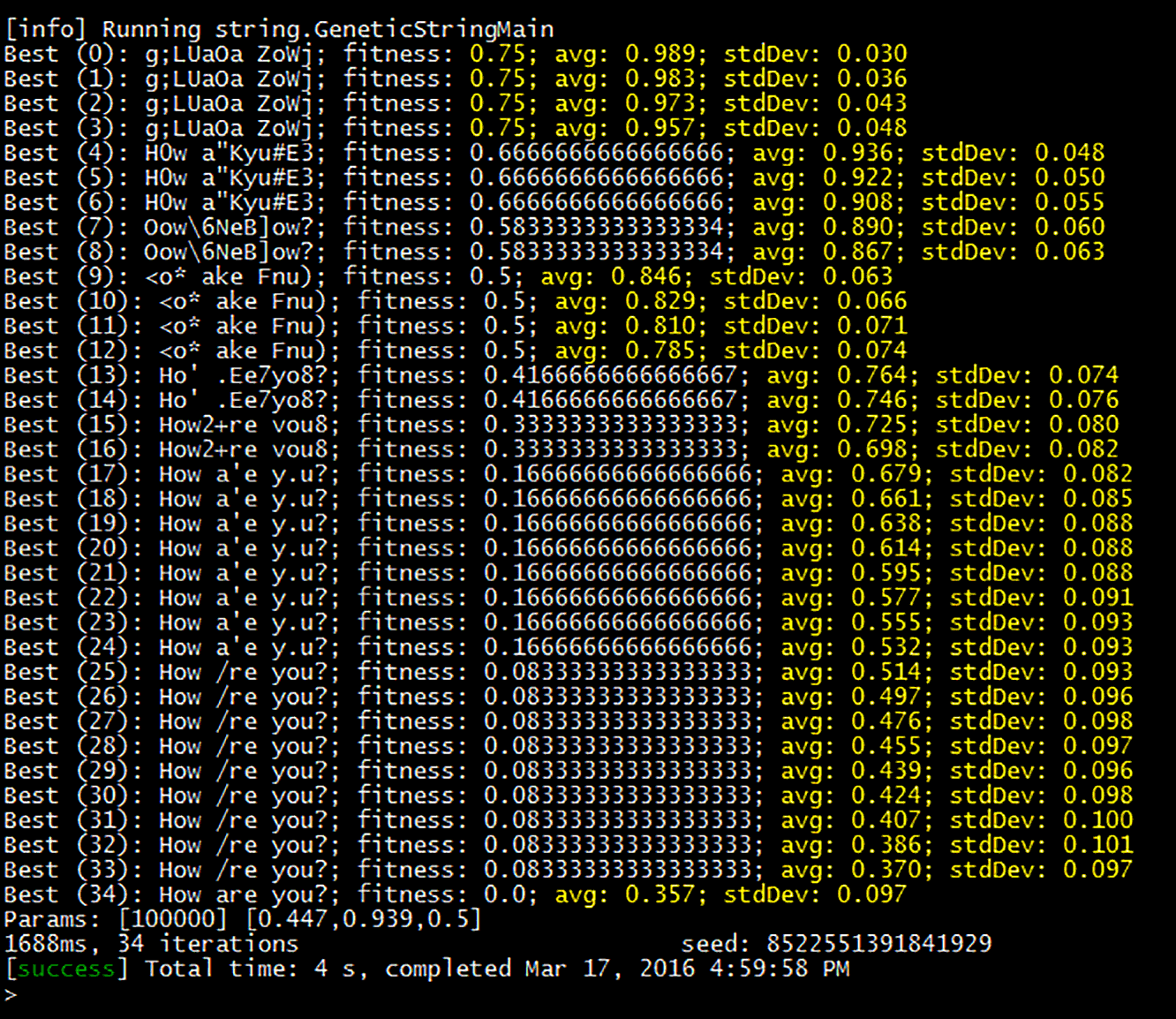
**חלק א – בעיות חיפוש לוקאלי**

**שאלה 1: (\*\*\*\*\*\*\*\* הערה: צריך איפשהו להסביר על הבעיה של ה strings \*\*\*\*\*\*\*)**

חישוב ממוצע – המנה של סכום ה fitness-ים חלקי כמות האיברים.

חישוב סטיית תקן – ממוצע ריבוע המרחקים מהממוצע.

דוגמאת הרצה: (חיפוש מחרוזת באלגוריתם גנטי)



**שאלה 2:**

הוספנו לבעיית **חיפוש המחרוזת** היוריסטיקה של "בול פגיעה" – הבאנו בונוס ב fitness למחרוזות שיש להן אות שנמצאת במחרוזת שמחפשים גם אם היא לא נמצאת שם.

קוד:

**public static double** heuristic3

(**char**[] elem, **char**[] target, **int** containsWeight, **int** eqWeight) {  
 **int** len = Math.*min*(elem.**length**, target.**length**);  
 **double** fitness = 0;  
 **for** (**int** i = 0; i < len; i++) {  
 **int** contains = *invIndicator*(*strContains*(target, elem[i]));  
 **int** eq = *invIndicator*(elem[i] == target[i]);  
  
 fitness += (**double**) (containsWeight \* contains + eqWeight \* eq) / (containsWeight + eqWeight);  
 }  
 **return** fitness / target.**length**;  
}

**public static int** invIndicator(**boolean** b) {  
 **return** b ? 0 : 1;  
}

היוריסטיקה זו לא שיפרה את ההיוריסטיקה המקורית – אדרבא – היוריסטיקה זו יוצרת Local Minima.

השוואה להיוריסטיקה הראשונה:

**שאלה 3:**

בעיית **מציאת מינימום לפונקציה**:

עלינו למצוא מינימום מוחלט לפונקציה בטווח מסויים.

מימוש על ידי אלגוריתם גנטי:

Gene:

ייצגנו מופע של הבעיה כנקודה במרחב.

Fitness:

הערך של הפונקציה בנקודה. עלינו למזער את ערך זה.

Mate:

Weighted Average – ממוצע ממושקל של הנקודות בכל מימד.

Mutate:

Noise – מזיז כל נקודה בערך רנדומי בטווח שנקבע מראש.

(\*\*\* למה מהם כדאי להוסיף קוד?? \*\*)

**שאלה 4:**

דגדגדג

**שאלה 5:**

בבעיית **חיפוש המחרוזת** – אסטרטגיות Mating:

Two Point Crossover – בחירת שני אינדקסים רנדומיים, לקיחת התוים שבינהם מהאב והשאר מהאם.

**public static char**[] twoPointCrossover(**char**[] x, **char**[] y, Random rand) {  
 **assert** x.**length** == y.**length**;  
  
 **int** len1 = rand.nextInt(x.**length**);  
 **int** len2 = rand.nextInt(x.**length** - len1);  
 **char**[] str = **new char**[x.**length**];  
 *arraycopy*(x, 0, str, 0, len1);  
 *arraycopy*(y, len1, str, len1, len2);  
 *arraycopy*(x, len1 + len2, str, len1 + len2, x.**length** - len1 - len2);  
 **return** str;  
}

Uniform Crossover – לקיחת כל תו שבאינדקס בהסתברות 0.5 מהאב ו- 0.5 מהאם.

**public static char**[] uniformCrossover(**char**[] x, **char**[] y, Random rand) {  
 **assert** x.**length** == y.**length**;  
  
 **char**[][] inputs = {x, y};  
 **char**[] str = **new char**[x.**length**];  
 **for** (**int** i = 0; i < x.**length**; i++) {  
 str[i] = inputs[rand.nextInt(2)][i];  
 }  
 **return** str;  
}

הערה: אסטרטגיות שיחלוף ומוטציות עבור בעיית N המלכות בשאלה 7 שבחלק ב'.

**שאלה 6:**

נרצה לבדוק את רגישות הפתרון לפרמטרים שונים שהוא תלוי בהם. הפעלנו אלגוריתם גנטי על הפרמטרים של האלגוריתם הגנטי לצורך מציאת פרמטרים אופטימאליים. חקרנו מה קורה לפרמטרים כאשר מזיזים אותם מאותה נקודה אופטימאלית, בנוסף לאסטרטגיות המוטציה והזיווג.

חיפוש מחרוזת:

Mate = One Point Crossover

אופטימאלי:

Population Size:

Mutation Rate:

Elitism Rate:

Top Ratio:

שינוי Population Size:

שינוי Mutation Rate:

שינוי Elitism Rate:

שינוי Top Ratio:

(\*\*\* לעשו גרפים לכל אחד מהם + לשכפל לכל mating + mutation. לעשות גם ל nQueens + מינימום לפונקציה \*\*\*)

אלגוריתם גנטי לחיפוש מחרוזת בבול פגיעה לעומת Hill climbing steepest ascent:

מימוש Hill climbing:

**def** hillClimbing(state: Array[Char]): String = {  
 **var** index = 0  
 **while**(heuristic(state) > 0 && index < state.length) {  
 **val** bestChar = *chars*.minBy(c => {  
 state(index) = c  
 **val** value = heuristic(state)  
 value  
 })  
 state(index) = bestChar  
 index += 1  
 }  
 state.mkString  
}

השוואת ביצועים:

כיצד ניתן להכליא בין שני האלגוריתמים?

ניתן להכליא בין אלגוריתם גנטי לאלגוריתם hill climbing על ידי

**חלק ב – בעיות עם אילוצים**

**שאלה 7:**

בעיית **N המלכות**:

עלינו למקם N מלכות על לוח שחמט בגודל NxN כך שאף מלכה אינה מאיימת על השנייה.

מימוש על ידי אלגוריתם גנטי:

Gene:

בהינתן גודל לוח N ייצגנו גן כלוח שח עם N מלכות עליו על ידי ייצוגו כפרמוטציה של מיקומי המלכות. מימשנו זאת על ידי מערך בגודל N כך שבאינדקס ה-i יהיה לנו מיקום השורה של המלכה שבעמודה ה-i-ית, כאשר אין שתי מלכות באותה שורה.

Fitness:

כמות ההתנגשויות של המלכות מנורמל בכמות התנגשויות מקסימאלית. דומה למה שראינו בשיעור.

Mate:

Partially Matched Crossover – כערך האב לאחר בחירת אינדקס אקראי, החלפתו בערך של האם, ואז שימור הפרמוטציה בהחלפה.

Ordered Crossover – העתקה של כמחצית מערכי הפרמוטציה של האב לבן, ולקיחת השאר מהאם תוך שימור הסדר.

Cycle Crossover – העתקת מחזור מעגלי שלם בפרמטוציה שמייצגים שני האבות לבן, ולקיחת השאר הפוכים.

Mutate:

Displacement – הזזה בפרמוטציה בכמות רנדומית של קטע רציף באורך רנדומי.

Exchange – החלפה של שני ערכים באינדקסים אקראיים בפרמוטציה.

Insertion – הזזה של ערך הנמצא באינדקס רנדומי כמות רנדומית של צעדים.

Simple Inversion – בחירת קטע רנדומי באורך רנדומי בפרמוטציה והיפוכו.

Inversion – בחירת קטע רנדומי, היפוכו, ואז הזזתו. הפרמטרים רנדומיים.

Scramble – בחירת קטע רנדומי באורך רנדומי ועירבולו.

מימוש על ידי Minimal Conflicts:

ייצוג חדש ללוח שח – כעת נייצג לוח בעל N מלכות עליו, כאשר בכל עמודה מלכה אחת, אך כעת נאפשר התנגשות של מלכות באותן שורות. מימוש – על ידי מערך בגודל N, כאשר באינדקס ה-i יהיה לי מספר השורה של הלכה בעמודה ה-i.

האלגוריתם:

נתחיל מלוח שח רנדומי. כל עוד קיימת מלכה שמאיימת על מלכה אחרת, נבחר מלכה רנדומית שמאיימת על מלכה אחרת, ונזיז אותה בעמודה שלה למיקום שבו כמות ההתנגשויות של הלוח החדש יהיה מינימאלי – אם יש כמה מיקומים כאלה – נבחר אחד מהם באופן רנדומי.

**public static** Optional<QueenBoard> minimalConflictsAlg

(QueenBoard startingBoard, **long** endTimeInNano, Random rnd){  
 **while**(System.*nanoTime*() < endTimeInNano){  
 List<Integer> conflictCols = startingBoard.getConflictCols();  
 **if** (conflictCols.size() == 0)  
 **return** Optional.*of*(startingBoard);  
 **int** col = conflictCols.get(rnd.nextInt(conflictCols.size()));  
 startingBoard.moveToBest(col, rnd);  
 }  
  
 **return** Optional.*empty*();  
}

כיצד ניתן להכליא בין האלגוריתם של Minimal Conflicts עם אלגוריתם גנטי?

ניתן להשתמש באיטרציה יחידה של minimal conflicts כפונקציית mutate עבור אלגוריתם גנטי. כלומר, באלגוריתם הגנטי המוטציה תיקח לוח, תבחר מלכה רנדומית שמאיימת על מלכה אחרת, ותזיז אותה לשורה שבה יהיה מספר התנגשויות מינימאלי בלוח.

מדוע זוהי הכלאה טובה? כיוון שאז המוטציה אכן תשפר את הגן, ותביא אותו למצב יותר טוב. במקום שמוטציה תשנה את הגן באופן רנדומי, נדע שהמוטציה הורידה את כמות ההתנגשויות של הלוח.

**שאלה 8:**

בעיית ה-**Knapsack**:

בהינתן גודל שק, ורשימה של חפצים עם משקלם ומחירם, עלינו לקחת כמות מכל חפץ כך שנמקסם את המחיר כאשר לא נחרוג מגודל השק. הבעיה הינה NP-Hard.

מימוש על ידי אלגוריתם גנטי:

Gene:

ביינתן מופע של הבעיה – גודל שק, מחרי פריטים ומשקלם, ייצגנו גן כמיפוי מכל פריט לכמות הפעמים שלקחנו אותו. מימשנו זאת כמערך כך שבאינדקס ה-i יש את כמות הפריטים שבפריט ה-i (הפריטים ממויינים לפי משקל)

Fitness:

חסמנו מלמעלה את המחיר המקסימאלי שניתן לקחת על ידי לקיחה של מקסימום מהמוצר עם יחס מחיר-משקל הכי גבוה. לקחנו את ההפרש של 1 עם היחס בין המחיר הנוכחי של השק ביחס לחסם העליון על המחיר. את גודל זה עלינו למזער.

**def** fitnessOfUppedBound(): Double = {  
 1 - totalValue() / instance.valueUpperBound  
}

Mate:

One Point Crossover – בחירת אינדקס רנדומי, לקיחת כמות הפריטים מההורה הראשון עד לאותו האינדקס, ואת השאר מההורה השני. אם חרגנו מגודל השק נקצץ פריטים באופן רנדומי עד שנחזור לגודל חוקי.

Two Point Crossover – בחירת שני אינדקסים רנדומיים, לקיחת כמות הפריטים מההורה הראשון שבין שני האינדקסים הללו. את כל השאר ניקח מההורה השני. אם חרגנו מגודל השק נקצץ פריטים באופן רנדומי עד שנחזור לגודל חוקי.

Uniform Crossover – עבור כל פריט ניקח את הכמות שהוא נילקח בסיכוי 50% מהאבא ובסיכוי 50% מהאמא. אם חרגנו מגודל השק נקצץ פריטים באופן רנדומי עד שנחזור לגודל חוקי.

Mutate:

One Point Mutate – בחירת פריט רנדומי, העלאת כמות הפעמים שלקחנו אותו באחד. אם חרגנו מגודל השק נקצץ פריטים באופן רנדומי עד שנחזור לגודל חוקי.

Binomial Mutate – כל פריט בהסתברות p ניקח ממנו עוד אחד. אם חרגנו מגודל השק נקצץ פריטים באופן רנדומי עד שנחזור לגודל חוקי.