מעבדה 2 בבינה מלאכותית

נושא:

אלגוריתמים גנטיים, ממטיים ואבולוציה

­­

מגישים:

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

מנחה:

מר שי בושנסקי



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

**חלק א – שכלול המנוע הגנטי**

**שאלה 1:**

הוספנו הדפסה של זמן הריצה של האלגוריתם: (\*\*\*\*\* האם שי ביקש הדפסה בכל דור??\*\*\*\*\*)

כמו כן, ב-JVM אין תמיכה בגישה לכמות נקישות השעון של המעבד, לפיכך לא יכולנו להוסיף תמיכה בכך.

דוגמאת הרצה: (חיפוש מחרוזת באלגוריתם גנטי)



**שאלה 2: (\*\*\*\*\*יובל לא יודע את המשמעות של הדברים, שאילן יכתוב\*\*\*\*\*\*\*)**

RWS:

דד (\*\*האם להוסיף שאילן לקח את הקוד ממאמר?\*\*)

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**;  
}

Scaling:

דדד

קוד:

SUS:

דדד

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;  
}

Ranking + Tournament:

דדד

קוד:

Aging Model:

דדד

קוד:

**שאלה 3:**

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

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 ימינה ולמעלה, רצינו מרחק שייצג את הפרמוטציה נאמנה.רצינו ייצוג של ה'דרך' שצריך כדי לעבור מלוח ללוח.

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 n מימדית). 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:**

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[A](stdDevThreshold : Double) **extends** LocalOptimaSignal[A]{  
 **override def** isInLocalOptima(population: Population[A]): Boolean = {  
 **val** fitnesses = population.*population*.map(\_.*fitness*)  
 **val** (avg, stdDev) = Util.*stdDev*(fitnesses) (*identity*)  
 stdDev < stdDevThreshold  
 }  
}

קריטריון דימיון פרטים לאבחון מינימום לוקאלי:

כאשר סטיית התקן של המרחקים מהפרט הטוב ביותר לשאר קטנה מסף מסויים – אבחנו שאנו נמצאים במינימום לוקאלי.

קוד:

**class** GeneSimilarityLocalOptimaDetector[A](distanceThresh : Double ,metric: Metric[A]) **extends** LocalOptimaSignal[A]{  
 **override def** isInLocalOptima(population: Population[A]): Boolean = {  
 **val** distances = **for** {  
 g1 <- population.*population* } **yield** metric.distance(population.*population*(0).*gene*, g1.*gene*)  
 **val** distanceStdDev = Util.*stdDev*(distances)(*identity*).\_2  
 distanceStdDev < distanceThresh  
 }  
}

**שאלה 5:**

Hyper mutation:

כאשר אובחן שאנו נמצאים במינימום מקומי, נגדיל את הסיכוי למוטציה.

Niching:

כאשר אובחן שאנו נמצאים במינימום מקומי, עבדנו עם פונקציית fitness חדשה שהענישה גנים הדומים לגנים אחרים.

קוד:

**public double** mapFitness(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);  
}

Random immigrants:

כאשר אובחן שאנו נמצאים במינימום מקומי, נוסיף לאוכלוסיה שלנו מספר גנים רנדומיים חדשים וטריים שירעננו את מאגר הגנים שלנו.

קוד:

מיד אחרי האליטיזם יבואו המהגרים הרנדומיים (במידת הצורך)

**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);  
}

**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.