מעבדה 1 בבינה מלאכותית

נושא:

מבוא לאלגוריתמים גנטיים

מגישים:

א) אילן גודיק, 316315332

ב) יובל אלפסי, 318401015

מנחה:

מר שי בושנסקי

תאריך הגשה:

18 למרץ 2016

תוכן עיניינים

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, מוטציות ורבייה בין גנים שונים. 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. (על ידי 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 שני 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 האוכלוסייה 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 האוכלוסייה, אחוז הElitism, אחוז האוכלוסייה 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 הפרמטרים שלו:

```
public abstract class GeneticMain<A> {
  public abstract Params defaultParams();
  public abstract int intsMax();
  public abstract GeneticAlg<A> alg(Params params, double maxTime);
}
```

אופטימיזציה של פרמטרים לאלגוריתם הגנטי

לצורך איתור הפרמטרים האידיאליים לזמן הריצה של האלגוריתמים הגנטיים שלנו, הפעלנו אלגוריתם גנטי על הפרמטרים של האגוריתם הגנטי לצורך מציאת אופטימום. ה-fitness – זמן הריצה של האלגוריתם הגנטי עם הפרמטרים המסויימים. זה היה תהליך מרתק ומסעיר. למדנו מהתוצאות האופטימליות שקיבלנו תכונות על האלגוריתם הספציפי ועל אופיו.

הגנים: מערכים של מספרים שלמים וממשיים

Fitness: אם הגיע לTimeout – לפי הfitness שאליו הגיע האלג' הגנטי מלמטה, fitness בין 0.5 ל1, ואם הגיע לפתרון – לפי אחוז זמן הריצה מתוך מגבלת הזמן – fitness בין 0 ל0.5.

מגבלת הזמן: קבוע k (אצלינו 4) כפול הזמן הטוב ביותר שנצפה עד כה.

זאת בכדי לתת Selection Pressure על הפתרונות הטובים, ולא לאבד זמן מהאלגוריתם על הפתרונות הרעים, וזאת בהתאמה דינאמית לאוכלוסיה שצפינו בה עד כה.

עצי חיפוש מונטה קרלו MCTS לאופטימיזציה ממשית

מתוך עניין, מימשנו אופטימיזציה למספרים חשבנו על מבנה מרחב חיפוש לשם אופטימיזציה ממשית (כגון לפרמטרים של אלגוריתם גנטי) בעזרת עצי חיפוש מונטה קרלו:

בכל שלב, הולכים הסתברותית לעלה בעץ באופן חכם: יש איזון בין Exploration – חיפוש תתי עצים עם פוטנציאל שעוד לא התגלה לExploitation – שיפור הפתרונות שאנו יודעים כי הם טובים. (יש קצת מתמטיקה בעניין)

כאשר מגיעים לעלה, אנחנו צריכים לעשות Rollout או Evaluation – במקרה שלנו, הרצת אלגוריתם גנטי עם הפרמטרים שמיוצגים ע"י הצומת הנוכחי של העץ.

טופולוגית העץ הבסיסית:

בשורש יש 0.5, תת עץ ימני מייצג את הטווח 0.5-1 והטווח השמאלי את 0-0.5.

פנייה מרובה לתת עץ מסויים מסמלת לאלגוריתם שתת עץ זה הוא טוב, ונחפש שם יותר את הפתרונות הטובים.

לשם יצירת וקטור של ממשיים, נשנה את טופולוגית העץ כך שברמה mod = i אנחנו עושים פיצול לאיבר הi בוקטור.

אלגוריתם זה הוא online – כלומר ניתן לבקש את התשובה הכי טובה שהוא הגיע אליה בכל רגע ורגע – וזאת מכיוון שאנו עושים סיבובים רבים קצרים על העץ, ומשפרים את הפתרון האופטימלי שלנו.

למציאת הפתרון האופטימלי, נלך לבן האופטימלי כל הזמן מהשורש עד לעלה, מכיוון שככל שרמת הצומת בעץ נמוכה יותר, הוא שיערוך בעל דיוק רב יותר של הפתרון.

את המנוע הגנטי שלנו מימשנו בשפה הפונקציונאלית Scala במשולב עם Java לצורך האצת ביצועים.

חלק א – בעיות חיפוש לוקאלי

שאלה 1:

בעיית **חיפוש מחרוזת**: עלינו למצוא מחרוזת סודית בהינתן היוריסטיקות שונות המתארות את המרחק למחרוזת זאת.

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

חישוב סטיית תקן – ממוצע ריבוע המרחקים מהממוצע.

דוגמאת הרצה: (חיפוש מחרוזת באלגוריתם גנטי)

```
[info] Running string.GeneticStringMain
Best (0): g;LUaOa ZoWj; fitness: 0.75; avg: 0.989; stdDev: 0.030
Best (1): g;LUaOa ZoWj; fitness: 0.75; avg: 0.983; stdDev: 0.036
Best (2): g;LUaOa ZoWj; fitness: 0.75; avg: 0.973; stdDev: 0.043
Best (3): g;LUaOa ZoWj; fitness: 0.75; avg: 0.957; stdDev: 0.048
Best (4): HOw a"Kyu#E3; fitness: 0.6666666666666666; avg: 0.936; stdDev: 0.048
Best (5): HOw a"Kyu#E3; fitness: 0.6666666666666666; avg: 0.922; stdDev: 0.050
Best (6): HOw a"Kyu#E3; fitness: 0.6666666666666666; avg: 0.908; stdDev: 0.055
Best (7): Oow\6NeB]ow?; fitness: 0.5833333333333334; avg: 0.890; stdDev: 0.060
Best (8): Oow\6NeB]ow?; fitness: 0.5833333333333334; avg: 0.867; stdDev: 0.063
Best (9): <o* ake Fnu); fitness: 0.5; avg: 0.846; stdDev: 0.063
Best (10): <o* ake Fnu); fitness: 0.5; avg: 0.829; stdDev: 0.066
Best (11): <o* ake Fnu); fitness: 0.5; avg: 0.810; stdDev: 0.071
Best (12): <o* ake Fnu); fitness: 0.5; avg: 0.785; stdDev: 0.074
Best (13): Ho' .Ee7yo8?; fitness: 0.4166666666666667; avg: 0.764; stdDev: 0.074
Best (14): Ho' .Ee7yo8?; fitness: 0.4166666666666667; avg: 0.746; stdDev: 0.076
Best (15): How2+re vou8; fitness: 0.3333333333333333; avg: 0.725; stdDev: 0.080
Best (16): How2+re vou8; fitness: 0.3333333333333333; avg: 0.698; stdDev: 0.082
Best (17): How a'e y.u?; fitness: 0.16666666666666666; avg: 0.679; stdDev: 0.082
Best (18): How a'e y.u?; fitness: 0.16666666666666666; avg: 0.661; stdDev: 0.085
Best (19): How a'e y.u?; fitness: 0.16666666666666666; avg: 0.638; stdDev: 0.088
Best (20): How a'e y.u?; fitness: 0.16666666666666666; avg: 0.614; stdDev: 0.088
Best (21): How a'e y.u?; fitness: 0.16666666666666666; avg: 0.595; stdDev: 0.088
Best (22): How a'e y.u?; fitness: 0.16666666666666666; avg: 0.577; stdDev: 0.091
Best (23): How a'e y.u?; fitness: 0.16666666666666666; avg: 0.555; stdDev: 0.093
Best (24): How a'e y.u?; fitness: 0.16666666666666666; avg: 0.532; stdDev: 0.093
Best (25): How /re you?; fitness: 0.08333333333333333; avg: 0.514; stdDev: 0.093
Best (26): How /re you?; fitness: 0.08333333333333333; avg: 0.497; stdDev: 0.096
Best (27): How /re you?; fitness: 0.08333333333333333; avg: 0.476; stdDev: 0.098
Best (28): How /re you?; fitness: 0.08333333333333333; avg: 0.455; stdDev: 0.097
Best (29): How /re you?; fitness: 0.08333333333333333; avg: 0.439; stdDev: 0.096
Best (30): How /re you?; fitness: 0.08333333333333333; avg: 0.424; stdDev: 0.098
Best (31): How /re you?; fitness: 0.08333333333333333; avg: 0.407; stdDev: 0.100
Best (32): How /re you?; fitness: 0.08333333333333333; avg: 0.386; stdDev: 0.101
Best (33): How /re you?; fitness: 0.08333333333333333; avg: 0.370; stdDev: 0.097
Best (34): How are you?; fitness: 0.0; avg: 0.357; stdDev: 0.097
Params: [100000] [0.447,0.939,0.5]
1688ms, 34 iterations                          seed: 8522551391841929
[success] Total time: 4 s, completed Mar 17, 2016 4:59:58 PM
>
```

שאלה 2:

הוספנו לבעיית **חיפוש המחרוזת** היוריסטיקה של "בול פגיעה" – ההיוריסטיקה מחשיבה תווים שנמצאים בדיוק באותו מיקום כמו במחרוזת הסודית וגם תווים שמוכלים במחרוזת הסודית אך לא נמצאים במקומם.

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. בנוסף, זמן החישוב החישוב של פונקציה זו יקר יותר מזמן החישוב של ההיוריסטיקה הראשונה. השתמשנו בהיוריסטיקה נוספת שספרה אך ורק התאמות מדויקות של מיקומים של תווים. היוריסטיקה זו התגלתה כמתאימה ביותר לבעיה, היא הגיעה לפתרון בזמן מינימאלי מבין כל שאר ההיוריסטיקות האחרות שניתנו, תוך יציבות מעולה והימנעות ממינימות לוקאליות.

השוואה להיוריסטיקה הראשונה – בחוברת ניתוח הפרמטרים שבסוף הדו"ח.

שאלה 3:

בעיית **מציאת מינימום לפונקציה**:

עלינו למצוא מינימום מוחלט לפונקציה בטווח מסויים.

מימוש על ידי אלגוריתם גנטי:

Gene:

ייצגנו מופע של הבעיה כזוג מספרים ממשיים.

Fitness:

הערך של הפונקציה בנקודה. עלינו למזער את ערך זה.

Mate:

Weighted Average – ממוצע ממושקל של שתי הנקודות בשני המימדים. המשקל של הממוצע נבחר באופן אקראי

```
def mate(x: Func, y: Func): Func = {
  Func(randAvg(x.x1, y.x1, rand), randAvg(x.x2, y.x2, rand))
}

def randAvg(x: Double, y: Double, rand: Random): Double = {
  val w = rand.nextDouble()
  x * w + y * (1 - w)
}
```

Mutate:

Noise – מזיז כל ערך במרחק אקראי d בתחום $[-mutationSize, mutationSize]$ שמועבר כפרמטר לאלגוריתם הגנטי.

```
def mutate(a: Func): Func = {
  val delta1 = (rand.nextFloat() - 0.5) * MutationSize * 2
  val delta2 = (rand.nextFloat() - 0.5) * MutationSize * 2
  Func(a.x1 + delta1 max 0 min 1, a.x2 + delta2 max 0 min 1)
}
```

שאלה 4:

אסטרטגיית בחירה של **טורניר**: באסטרטגיה זו, האוסף ממנו יבחר זוג ההורים הוא תת קבוצה אקראית של האוכלוסיה בגודל k, מתוכה יבחר את הגן הכי טוב בהסתברות p, את השני הכי טוב בהסתברות p(1-p) וכן הלאה.

מימוש אסטרטגיה זו:

```
public <A> A chooseParent(Population<A> parentsPool, Random rand) {
    int i = 0;
    while(true) {
        if(i == tournamentSize)
                 i = 0;
        else if(rand.nextFloat() < chooseBestProbability)
                 return parentsPool.population[i].gene;
        else
                 i++;
    }
}

public <A> void populateParentsPool
   (Population<A> population, Population<A> parentsPool, Random rand) {
    assert parentsPool.population.length == tournamentSize;
    int popSize = population.population.length;
    for (int i = 0; i < parentsPool.population.length; i++) {
        parentsPool.population[i] =
             population.population[rand.nextInt(popSize)];
    }
    JavaUtil.sortGenes(parentsPool.population);
}
```

ככל ש-k יותר גדול כך ה- Selection pressure יותר גבוה – זאת כיוון שבחירה זו שואפת להיות בחירה מתוך כל האוכלוסיה, מתוכה נבחר בהסתברות גבוהה את הטובים ביותר.

לעומת זאת, ככל ש-k יותר נמוך כך ה- Diversity יותר גבוה ויש יותר מקודם לגנים חלשים שאולי יתרמו מתכונותיהם הנדירות.

דעתינו על האסטרטגיה: ראשית, הטובים ביותר לאו דווקא יתרבו, יש צ'אנס סביר שלגנים טובים במיוחד לא יצא להזדווג ולהעביר לדור הבא את התכונות המשובחות שלהם. שנית, מבחינת ביצועים, שיטת הטורניר בעייתית מבחינת כמות הזמן אותה היא צורכת עבור בחירת k איברים.

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;
}
```

סטטיסטיקות וביצועים בחוברת ניתוח הביצועים שבסוף.

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. אחד הפרמטרים החשובים ביותר לנו הוא היציבות של האלגוריתם. לא נרצה שפעם בכמה הרצות נקבל שהאלגוריתם ירוץ זמן רב מדי. לפיכך, לקחנו זאת בחשבון בבדיקת הביצועים שלנו. כל מופע של פרמטרים הרצנו פעמים רבות.

סטטיסטיקות – מפורטות בחוברת ניתוח הפרמטרים שבסוף הדו"ח.

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

אלגוריתם Hill Climbing הינו אלגוריתם לחיפוש במרחב מצבים בהינתן ידע חיצוני (היוריסטיקה). האלגוריתם מתחיל ממצב התחלתי ממנו הוא מתקדם בכל שלב לשכן בעל ההיוריסטיקה הטובה ביותר. אלגוריתם Hill Climbing רגיש ביותר ל- local minima ולפיכך אינו שימושי עבור היוריסטיקות שאינן מושלמות.

במקרה שלנו במחרוזות:

- כל מצב מתאר מחרוזת באורך הקטן או שווה מהמחרוזת הסודית.
- המצב ההתחלתי הוא מחרוזת ריקה.
- שכנים של מצב הם כל המחרוזות באורך גדול באחד כך שהרישא של השכן שווה למחרוזת המיוצגת על ידי הצומת.
- יש בידינו היוריסטיקה מושלמת, מספר ההתאמות שבמחרוזת.

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
}
```

<u>ביצועים:</u> 0.11 ms בממוצע למחרוזת באורך 30, לעומת 2.78 ms בממוצע לאלגוריתם גנטי.

רעיון להכלאה בין אלגוריתם גנטי לאלגוריתם hill climbing: במקום בכל שלב לבחור את השכן הכי טוב, נפעיל אלגוריתם גנטי קטן על אוכלסיית השכנים, אותה נפתח מעט, ומהם נבחר את הכי טוב, וחוזר חלילה.

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();
}
```

זמן ריצה: 33.5 ms בממוצע ללוח בגודל 10, לעומת בסביבות 2 ms לאלגוריתם גנטי.

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:

בהינתן 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 היחס המקסימאלי של מחיר-משקל כפול גודל השק. כ- fitness 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.

מימוש:

הפונקציה trim – מקצצת פריטים מהשק עד שנחזור לסכום חוקי

```
def isValid(): Boolean = totalWeight() < instance.capacity

def trim(rnd: Random): Unit = {
  while (!this.isValid) {
    val index = rnd.nextInt(amounts.length)
    if (amounts(index) > 0)
      amounts(index) = amounts(index) - 1
  }
}
```

הפונקציה binomial mutate:

```
public static void binomialMutate
   (double mutateProb, KnapsackElement instance, Random rand){
    for (int i = 0; i < instance.amounts().length; i++)
        if (rand.nextDouble() < mutateProb)
            instance.amounts()[i]++;
    instance.trim(rand);
}
```