**פתרון Multi-Dimensional Knapsack בעזרת המנוע הגנטי**

**ייצוג הבעייה:** BitSet

לשם ייצוג פתרון לבעייה, מימשנו BitSet המייצג את החפצים שנבחרו.

כל הפעולות הגנטיות מבוצעות על BitSet זה.

הBitSet מיוצג ע"י מערך של Longים באורך 64 ביטים כל אחד.

**Mating:** One Point Crossover

מימשנו זאת בין BitSets ע"י Mask בבלוק של נקודת החציה ולקיחת הביטים המתאימים מx ומy, העתקת הבלוקים לפני נקודת החצייה מx והבלוקים אחרי נקודת החצייה מy, ולאחר מכן Trim.

**Mutation:**

מוטציה של ביט במיקום אקראי בBitSet, ולאחר מכן תיקון התוצאה – Trim.

**איבר אקראי:**

ייצוג Longים אקראיים (זוהי הפעולה הפרימיטיבית בPRNG שאחנו משתמשים בו: XorShift128+), וניקוי הביטים בבלוק האחרון, בהתאם לכמות הבלוקים המתבקשת. לאחר מכן מבוצע Trim.

**Trim:**

כל הפעולות הגנטיות יכולות לייצר גנים לא תקינים – שלא מקיימים את התנאים של כל השקים.

לשם כך, אנו עושים לאחר כל פעולה גנטית את הפעולה Trim:

* הוצא איבר אקראי מהפתרון.
* אם הפתרון תקין בשק הנוכחי, עבור לשק הבא.

בכך אנו משיגים שכל החפצים נכנסים לתוך כל השקים.

**Fitness:**

**מטריקה:** Hamming Distance

ממומש ע"י PopCount (ספירת הביטים הדולקים) על הXOR של שתי הBitSets.

**Taken Items:**

מעבר לייצוג דואלי – מערך של כל האינדקסים של האיברים שנלקחו לפתרון.

ממומש ע"י lowestBitSet מהיר על הBitSet.

בנוסף, על מנת להימנע מהקצאות, אנו תמיד רושמים את התוצאה לתוך IntBuffer יחיד לכל ריצה / Thread.

המעבר לייצוג זה קריטי לשם פעולה יעילה של Trim – בחירת איבר אקראי מתוך כל האיברים שנבחרו לפתרון.

בנוסף זה תורם לחישוב קל של הערך הכולל של פתרון.

**מימוש:**

נציג את המימוש כמעט בכללותו, על מנת להדגים את כל מה שנדרש על מנת להוסיף בעיה חדשה למנוע הגנטי.

1. **ייצוג הבעייה:**

**case class** Sack(capacity: Int, itemWeights: Array[Int])

**case class** MDKnapsackInstance(name: String, values: Array[Int],

sacks: Array[Sack], optimum: Int)

ומימוש של BitSet לשם ייצוג פתרונות אפשריים לבעייה:

**public final class** BitSet **implements** Serializable, Cloneable {

**private final long**[] **bits**;  
 **public final int numBits**;  
  
 **public** BitSet(**int** numBits) {  
 **this**.**numBits** = numBits;  
 **int** numLongs = numBits >>> 6;  
 **if** ((numBits & 0x3F) != 0) {  
 numLongs++;  
 }  
 **bits** = **new long**[numLongs];  
 }  
  
 **public** BitSet(**int** numBits, **long**[] bits) {  
 **this**.**numBits** = numBits;  
 **this**.**bits** = bits;  
 }  
  
 **public boolean** get(**int** index) {**return** (**bits**[index >>> 6] & 1L << (index & 0x3F)) != 0L;  
 }  
  
 **public void** set(**int** index) {**bits**[index >>> 6] |= 1L << (index & 0x3F);  
 }  
  
 **public void** clear(**int** index) {**bits**[index >>> 6] &= ~(1L << (index & 0x3F));  
 }  
}

1. **Genetic:**

מימוש כל הפעולות הגנטיות.

**class** GeneticMDKnapsack(instance: MDKnapsackInstance, rand: Random)

**extends** Genetic[BitSet] {

**val** *itemsBuffer* =  
 **new** IntBuffer(instance.values.length)  
**override def** fitness(gene: BitSet): Double = {  
 1 - instance.value(gene, *itemsBuffer*).toDouble / instance.optimum  
 }  
**override def** score(gene: Gene[BitSet]): Double = {  
 instance.value(gene.*gene*, *itemsBuffer*)  
 }  
**override def** randomElement(rand: Random): BitSet = {  
 **val** items = BitSet.*randomBitSet*(instance.values.length, rand)  
 instance.trim(items, *itemsBuffer*, rand)  
 }  
**override def** mate(x: BitSet, y: BitSet): BitSet = {  
 **val** offspring = MDKnapsack.*mate*(x, y, instance, rand)  
 instance.trim(offspring, *itemsBuffer*, rand)  
 }  
**override def** mutate(items: BitSet): BitSet = {  
 **val** i = rand.nextInt(instance.values.length)  
 items.set(i)  
 instance.trim(items, *itemsBuffer*, rand)  
 }  
  
 **override def** metric(): Metric[BitSet] = **new** Metric[BitSet] {  
 **override def** distance(x: BitSet, y: BitSet): Double = {  
 BitSet.*hammingDistance*(x, y).toDouble / x.*numBits* }  
 }  
}

**מימושי הפעולות הגנטיות בפועל:**

**public static** BitSet randomBitSet(**int** numBits, Random rand) {  
 **int** numLongs = numBits >>> 6;  
 **if** ((numBits & 0x3F) != 0) {  
 numLongs++;  
 }  
 **long**[] bits = **new long**[numLongs];  
 **for** (**int** i = 0; i < numLongs; i++) {  
 bits[i] = rand.nextLong();  
 }  
 *// Clear all the irrelevant bits in the last block.  
 // For canonicity of representation.* **if** ((numBits & 0x3F) != 0) {  
 bits[numLongs - 1] &= (1L << (numBits & 0x3F)) - 1L;  
 }  
 **return new** BitSet(numBits, bits);  
}

**public int** lowestBit() {  
 **int** lowestBit = 0;  
 **for** (**int** i = 0; i < **bits**.**length**; i++) {  
 **int** index = Long.*numberOfTrailingZeros*(**bits**[i]);  
 lowestBit += index;  
 **if** (index != 64 && lowestBit < **numBits**) **return** lowestBit;  
 }  
 **return** -1;  
}  
  
*// Crossover operation, @i being the number of bits of @x to be copied.*

*// The rest are from @y.***public static** BitSet crossOver(BitSet x, BitSet y, **int** i) {  
 **if**(x.**numBits** != y.**numBits**) {  
 **throw new** IllegalArgumentException(**"The BitSets must have the same size"**);  
 }  
 **int** crossoverBlock = i >>> 6;  
 **int** crossoverIndex = i & 0x1F;  
 **long** bx = x.**bits**[crossoverBlock];  
 **long** by = y.**bits**[crossoverBlock];  
 **long** crossoverMask = (1 << crossoverIndex) - 1;  
 **long** newCrossoverBlock = (crossoverMask & bx) | (~crossoverMask & by);  
 **long**[] newBits = **new long**[x.**bits**.**length**];  
 System.*arraycopy*(x.**bits**, 0, newBits, 0, crossoverBlock);  
 newBits[crossoverBlock] = newCrossoverBlock;  
 System.*arraycopy*(y.**bits**, crossoverBlock + 1,

newBits, crossoverBlock + 1, y.**bits**.**length** - (crossoverBlock + 1));  
 **return new** BitSet(x.**numBits**, newBits);  
}  
  
**public static int** hammingDistance(BitSet x, BitSet y) {  
 **if**(x.**numBits** != y.**numBits**) {  
 **throw new** IllegalArgumentException(**"The BitSets must have the same size"**);  
 }  
 **int** distance = 0;  
 **for** (**int** i = 0; i < x.**bits**.**length**; i++) {  
 distance += Long.*bitCount*(x.**bits**[i] ^ y.**bits**[i]);  
 }  
 **return** distance;  
}

**מימוש הפעולות: Trim & Taken Items**

**public static void** takenItems(BitSet items, IntBuffer takenItemsBuffer) {  
 takenItemsBuffer.clear();  
 BitSet cloned\_items = items.clone();  
 **int** i = cloned\_items.lowestBit();  
 **while** (i != -1) {  
 takenItemsBuffer.add(i);  
 cloned\_items.clear(i);  
 i = cloned\_items.lowestBit();  
 }  
}

**public static void** trim(BitSet items, Sack[] sacks,

IntBuffer takenItemsBuffer,

Random rand) {  
 *// Trim to each sack at a time.* **for** (Sack sack : sacks) {  
 *// Inside, because it changes,*

*// and weightOfItems requires the correct one.  
 takenItems*(items, takenItemsBuffer);  
  
 **int**[] itemWeights = sack.itemWeights();  
 *// Contains the correct values,*

*// because we just calculated them w/ takenItems.*

**int** weightInSack = *weightOfItems*(takenItemsBuffer, itemWeights);

**int** sackCapacity = sack.capacity();  
 **while** (weightInSack > sackCapacity) {  
 **int** dropIndex =

takenItemsBuffer.get(rand.nextInt(takenItemsBuffer.size()));  
 **if** (items.get(dropIndex)) {  
 items.clear(dropIndex);  
 weightInSack -= itemWeights[dropIndex];  
 }  
 }  
 }  
}

1. **Genetic Metadata:**

לשם הוספת בעיה חדשה למנוע הגנטי יש לתת את הMetadata לבעייה, הכוללת את שם הבעיה, וערכי ברירת מחדל (אפשר גם להשמיט ולקבל את הערכים הרגילים, חובה לממש רק שם ואת הייצוג הפרמטרי לבעייה בgenetic).

**class** MDKnapsackMetadata(instance:MDKnapsackInstance) **extends** GeneticMetadata[BitSet]{   
 **override def** name: String = **"Multi-Dimensional Knapsack"  
  
 override def** defaultMaxTime: Double = 2.0  
 **override def** defaultPrintEvery: Int = 1000  
  
 **override def** genetic: Parametric[Genetic[BitSet]] =  
 Parametric.*point* {  
 **new** GeneticMDKnapsack(instance, *rand*)  
 }  
  
 *// To be overwritten to provide problem-specific defaults.* **override def** intNamesDefaults: Map[String, Int] = *Map*(  
 **"Population Size"** -> 216  
 )  
  
  
 **override def** intsNamesMax: Map[String, Int] = *Map*(  
 **"Population Size"** -> 512  
 )  
  
 **override def** doubleNamesDefaults: Map[String, Double] = *Map*(  
 **"Elitism Rate"** -> 0.34,  
 **"Gene Similarity Threshold"** -> 0.025,  
 **"Local Optimum: Elitism Rate"** -> 0.525,  
 **"Local Optimum: Hyper Mutation Rate"** -> 0.52,  
 **"Local Optimum: Immigrants Rate"** -> 0.074,  
 **"Local Optimum: Top Ratio"** -> 0.064,  
 **"Mutation Rate"** -> 0.624,  
 **"Top Ratio"** -> 0.988  
 )

*// Use Deduplication by default for MD-Knapsack.* **override def** defaultEngine: Parametric[GeneticEngine] = {  
 **val** normalGeneration = **for** {  
 selectionStrategy <- *topSelection* mutationStrategy <- *mutation* elitism <- *elitism* } **yield new** Generation(selectionStrategy, mutationStrategy,

*Array*(elitism), **new** DeduplicatedConstruction, *Array*())  
  
 **val** localOptimaGeneration = **for** {  
 selectionStrategy <- *topSelection* mutationStrategy <- *hyperMutation* elitism <- *elitism* immigrants <- *randomImmigrantsElitism* } **yield new** Generation(selectionStrategy, mutationStrategy,

*Array*(elitism, immigrants), **new** DeduplicatedConstruction, *Array*())  
  
 *geneticEngine*(*geneSimilarity*, normalGeneration, localOptimaGeneration)  
 }  
}

**שיפור המנוע הגנטי**

**Survival Selection:**

כעת ניתן לבחור כל תת קבוצה של .

ממומש כנגד ממשק של "שים גן חדש/ישן בדור הבא"

וזה מאפשר:

**Deduplication of Genes:**

מניעת כפילויות של גנים באוכלוסייה.

דבר זה היה מאוד קריטי עבור תוצאות טובות בMD-Knapsack, כמפורט באנליזה של MD-Knapsack, ללא Deduplication אנחנו לא מצליחים להגיע לפתרונות האופטימליים רוב הזמן.

**בחירת מנוע ברירת מחדל Per בעייה:**

אפשרנו לבחור מנוע גנטי ברירת מחדל הספציפי לבעייה.

לדוגמא עבור MD-Knapsack בחרנו להוסיף Deduplication, ובBaldwin בחרנו מנוע פשוט ללא Features נוספים כגון התמודדות עם מינימום לוקאלי ועוד, בכדי לשחזר את הניסוי המקורי כמה שיותר.

**Meta Genetic Algorithm:**

עבור בעיית הMD-Knapsack היה מאוד משמעותי היכולת של האלגוריתם המטא-גנטי לעשות אופטימיזציה גם כאשר לא מגיעים לפתרון אופטימלי:

אם הגענו לפתרון אופטימלי, הFitness יהיה בטווח של בהתאם לכמות הזמן שלקח להגיע לפתרון האופטימלי.

אם לא הגענו לפתרון האופטימלי, הFitness יהיה בטווח של בהתאם לFitness שהאלגוריתם הגנטי מלמטה הצליח להגיע אליו בתום מכסת הזמן שלו.

בכך, האלגוריתם המטא-גנטי יכול לפעול למען שיפור הFitness שניתן להגיע אליו, ובסופו של דבר מצליח להגיע לאופטימלי ולעבור את סף ה0.5 הקריטי, ולשפר זמן התכנסות משם.

**רעיונות להמשך**

* ניתן לעשות Monte Carlo Tree Search על הMD-Knapsack, על מנת לעשות חיפוש חכם יותר על מרחב האפשרויות במקום מעבר חסר ידע על העץ, וזה בצירוף הCutoffs שעשינו בBranch&Bound.
* אנו חושבים שSimulated Annealing יכולה להיות גישה טובה מאוד לשם האלגוריתם הMeta Genetic.

**MD-Knapsack Analysis**

**עם Deduplication:**

* מצאנו את האופטימום בכל הבעיות
* אך לא בכל ההרצות האינדיבידואליות של האלגוריתם הגנטי

רעיון הנובע מכך:

* ניתן להריץ כמה 'נישות' של האלגוריתם הגנטי – כמה הרצות נפרדות של האלגוריתם הגנטי, שניתן לקדם כל אחת מהן בדורות בנפרד (אפשר גם במקביל), ובכך להעלות את הסיכוי שבמכסת זמן ספציפית נקבל את הפתרון האופטימלי אם התכנסנו לאוכלוסייה טובה מתוך האקראיות לפחות באחד מההרצות.
* בנוסף ניתן לשתף גנים בין נישות וכו'.

אחוזי הצלחה:

* ל 41/55 מהבעיות יש 100% הצלחה – מגיעים לאופטימום בכל ההרצות.
* ל 7/55 מהבעיות יש >90% הצלחה.
* הבעיות הכי בעייתיות:

- SENTO2, HP2: 75% אחוזי הצלחה

- WEING7: 60% אחוזי הצלחה

- PET7: 45% אחוזי הצלחה

זמני ריצה בהרצות מוצלחות: (זמן התכנסות מקסימלי)

* עבור 34/55 מהבעיות: < 80 ms
* עבור 46/55 מהבעיות: < 180 ms
* לכל היותר 550 ms עבור הבעיות הבעייתיות

כאשר לא הגענו לפתרון האופטימלי

* הרוב המוחלט של הבעיות במרחק של פחות מ%0.5 מהאופטימלי
* לכל היותר במרחק של 2% מהאופטימלי
* 0.01% מרחק מהאופטימלי עבור WEING7

**ללא Deduplication:**

ישנן הרבה פחות בעיות עבורן משיגים 100% הצלחה.

בעיות רבות בטווח של 30%-60%

הבעיות הקשות, SENTO2, PET7, WEING7 הם באיזור של 1%-10% הצלחה.

**הדבר נובע מכך** שהגן של המינימום הלוקאלי מתחיל להתרבות, ונוצר מצב שהוא משתלט על רוב האוכלוסייה העליונה, ולא נותן הזדמנות בכלל לגנים אחרים להיכנס לרביה.