מעבדה 3 בבינה מלאכותית

נושא:

Multi-Dimentional 0-1 Knapsack

רלקסציה של תכנות לינארי

מגישים:

א) יובל אלפסי, 318401015

ב) אילן גודיק, 316315332

מנחה:

מר שי בושינסקי

תאריך הגשה:

22 לאפריל 2016

תוכן עיניינים

Multi-Dimentional 0-1 Knapsack

עלינו לפתור את בעיית השק הרב מימדית.

נתונים מספר שקים ומספר מוצרים. לכל שק ישנה קיבולת ולכל מוצר יש את ערכו ואת משקלו בכל שק. עבור כל עצם עלינו להחליט האם ניקח אותו או לא. אם ניקח אותו – הוא יתפוס מקום בכל השקים, בהתאם למשקלו בכל שק. עלינו למקסם את סכום ערכי המוצרים שלקחנו בעוד שלא חרגנו מקיבולות כל השקים.

הפיתרון הטריוויאלי לבעיה – ניסיון כל האופציות האפשריות לבחירה. סיבוכיות זמן – אקספוננציאלית.

מדובר בבעייה NP קשה.

במעבדה זו נסקור מספר גישות לפתרון הבעייה ונציע רעיונות נוספים לפתרונה.

1. פתרון הבעיה בעזרת Branch & Bound

אפשר למדל את הבעייה כמרחב חיפוש עם בחירות בינאריות בכל שלב – האם לקחת את החפץ או לא.

מעל מרחב זה אפשר להפעיל שיטות חיפוש, בנוסף לPruning של תתי עצים.

מימשנו את אלגוריתם הBranch & Bound בשפה #F, ופירוט נוסף יבוא בהמשך.

2. פתרון הבעיה בעזרת אלגוריתם גנטי

אפשר למדל את הבעיה כחיפוש מחרוזת בינארית הממקסמת ערך Fitness – הערך הכולל של החפצים שנבחרו.

אנחנו הוספנו את בעייה זו לGenetic Framework שבנינו במהלך שתי המעבדות הקודמות, בScala + Java.

Branch and Bound for MD-Knapsack

השתמשנו באלגוריתם בגישת הBranch and Bound לפתרון בעיית הMD-Knapsack כפי שהוצג במעבדה עם שי.

אלגוריתם זה הוא אלגוריתם חיפוש במרחב מצבים בצורת עץ, אך בנוסף לשיטת חיפוש רגילה על העץ, אנחנו שומרים על הפתרון הטוב ביותר שהגענו אליו עד כה, נותנים חסם עליון לכל תת עץ שעוברים עליו, ואם חסם עליון זה קטן מערכו של פתרון שהגענו אליו כבר, נקטום את תת עץ זה, ולא נחפש שם.

לשם גיזום מוקדם יותר, אנחנו מחשבים את הערך גם של צמתים פנימיים ללא החפצים שעוד לא בחרנו אם לקחת או לא, ומחשיבים אותם כפתרונות אפשריים.

```
member x.ShouldPrune (bestPrice : int, prunning : Solution -> int) : bool =
        prunning x < bestPrice
```

מבנה מרחב החיפוש:

כל צומת פנימי בעץ מייצג פתרון חלקי לבעיה – אילו חפצים לקחנו ואילו לא.

ברמה הi בעץ, נבחר האם לקחת את החפץ הi או לא – שמאלה מייצג לקחת, וימין מייצג לא לקחת.

ייצוג הבעייה:

```
type Knapsack(capacity : int)

type Item(price : int, constraints : Dictionary<Knapsack, int>)

                                    // Weight in each sack

type KnapsackProblem(name : string, items : Item array,
                     knapsacks : Knapsack array, optimal : int)

type Solution(itemsTaken : BitArray, prob : KnapsackProblem, openBits : int)
```

בדיקת תקינות פתרון: האם משקלי כל החפצים אינם חורגים מקיבולות כל השקים.

```
member x.IsValid =
  knapsacks |> Array.forall (
    fun knapsack ->
      let mutable sumOfConstraints = 0
         for i = 0 to items.Length - 1 do
           if itemsTaken.[i] then
            sumOfConstraints <- sumOfConstraints + items.[i].ConstraintOf knapsack
      sumOfConstraints < knapsack.Capacity)
```

היוריסטיקות לחסמים עליונים:

1. שק לא חסום

עבור צומת פנימי בעץ ברמה הi – החלטנו עד כה אילו מi האיברים הראשונים לוקחים ואילו לא, ועבור שאר האיברים, שעוד לא החלטנו עבורם, היוריסטיקה זו מניחה שקיבולת השק אינסופית, ולכן החסם העליון יהיה הערך שמתקבל אם כן ניקח את כל שאר האיברים שעוד לא החלטנו לגביהם, בנוסף לאיברים שכבר לקחנו.
אם חסם עליון זה נמוך מתוצאה שכבר קיבלנו, נקטום את תת העץ.

```
let unboundedKnapsackUpperBound (sol : Solution) =
        let mutable restPotentialPrice = 0
        for i = sol.OpenBits to sol.Prob.Items.Length - 1 do
            restPotentialPrice <- restPotentialPrice + sol.Prob.Items.[i].Price
        sol.Price + restPotentialPrice
```

2. שק חסום עם מילוי שברי

חסם זה הוא חסם עליון יותר הדוק מאשר "שק לא חסום", והוא פועל באופן הבא:

לכל שק, נמיין את כל החפצים ע"י ה'צפיפות' שלהם:

$$density = \frac{value}{weight}$$

נבחר את כל החפצים לפי סדר הצפיפות שלהם, כל עוד יש עוד מקום בשק,

ועבור החפץ האחרון, שלא נכנס לשק, ההיוריסטיקה תבחר "חלק שברי" ממנו:

$$density \cdot remainingWeight$$

זהו חסם עליון, מכיוון שתמיד כדאי יותר לקחת את החפצים שיש להם צפיפות גבוהה יותר, אם לא הייתה לנו המגבלה של בחירת חפצים שלמים.

בנוסף, כאשר יש לנו פתרון עם התאמה מדוייקת של חפצים שלמים, היוריסטיקה זו מהווה גם חסם תחתון לבעיה.

אפוא, כל שק ניתן למלא עד למחיר מקסימאלי כלשהו, כל מחיר כזה מהווה חסם עליון.
ניקח את חסם עליון מינימאלי, מכיוון שאם לא ניתן להשיג ערך טוב ממה שהשגנו עד כה בגלל אחד השקים, לא ניתן יהיה להשיג אותו בפרט עם יותר מגבלות, של שאר השקים.

```
let upperBoundFractional (sol : Solution) : int =
        sol.Prob.Knapsacks
        |> Seq.ofArray
        |> Seq.map (fun k -> fractionedFilledKnapsack sol k)
        |> Seq.min
```

קוד למציאת חלק שברי

```
let fractionedFilledKnapsack (sol : Solution) (knapsack : Knapsack) : int =
  let openedBits = sol.OpenBits
  let items = sol.Prob.Items
              |> Array.skip openedBits
              |> Array.sortBy (fun item ->
                 (float32 <| item.ConstraintOf knapsack) / (float32 item.Price))
             // Descending order of density.
  let length = items.Length
  let mutable index = 0
  let mutable filled = 0
  let mutable totalPrice = 0
  for i = 0 to openedBits - 1 do
      if sol.ItemsTaken.[i] then
         filled <- filled + sol.Prob.Items.[i].ConstraintOf knapsack
  while index < length && filled < knapsack.Capacity do
    let constr = items.[index].ConstraintOf knapsack
    // Fractional Part
    if filled + constr > knapsack.Capacity then
      let price = float32 <| items.[index].Price
      let rest = float32 <| knapsack.Capacity - filled
      let percent = rest / (float32 constr)
      totalPrice <- totalPrice + int32 (percent * price)
      index <- length
    // Whole Part
    else
      filled <- filled + constr
      totalPrice <- totalPrice + items.[index].Price
      index <- index + 1
  sol.Price + totalPrice
```

שיטות חיפוש על העץ

1. DFS – חיפוש לעומק

ישנה עדיפות להתקדם קודם לכיוון ה"לקחת מוצר" מאשר לכיוון ה"לא לקחת מוצר".

מימשנו בעזרת מחסנית, על מנת להימנע ממחסנית הקריאה לפונקציות, לשם שיפור ביצועים – הכנסת רק הדברים הרלוונטיים למחסנית.

```
let dfs upperBoundFunc (endTime : DateTime) (prob : KnapsackProblem) :
                      Solution * DateTime option =
  let startingSolution = Solution.Empty prob
  let mutable bestSolution = startingSolution
  let mutable maybeFindingTime : DateTime option = None
  let solutionsToBranch = new Stack<Solution>()
  solutionsToBranch.Push (startingSolution)
  while DateTime.Now < endTime && solutionsToBranch.Count > 0 do
    let sol = solutionsToBranch.Pop()
    let openedBits = sol.OpenBits
    let nextIndex = sol.OpenBits
    if sol.Price > bestSolution.Price then
       bestSolution <- sol
       if bestSolution.Price = sol.Prob.Optimal then
          maybeFindingTime <- Some(DateTime.Now)
    if openedBits < sol.Prob.Items.Length then
      let with1, with0 = sol.Branch
      if with0.IsValid && not <|
         with0.ShouldPrune (bestSolution.Price, upperBoundFunc)
      then solutionsToBranch.Push with0
      if with1.IsValid && not <|
         with1.ShouldPrune (bestSolution.Price, upperBoundFunc)
      then solutionsToBranch.Push with1
  (bestSolution, maybeFindingTime)
```

2. חיפוש Best First Search - BFS

נתחזק תור עדיפויות ממנו נפתח בכל שלב את הצומת בעל הFractional Upper Bound הטוב ביותר, ונכניס את ילדיו. גם כאן נגזום צמתים שערכם נמוך מהערך הטוב ביותר עד כה.

מכיוון שככל הנראה לא נגיע לעלים בדרך זו, אך צריך בכל מקרה לתת פתרונות טובים בשביל חסמים עליונים. לכן אנו רצים מכל צומת שפותחים עד לעלה, באופן חמדני, ואת צומת זה נבחן – האם הוא הטוב ביותר עד כה. שיטה זו נתנה שיפור משמעותי על גבי DFS פשוט.

השתמשנו בערימה בינארית לצורך תור העדיפויות. השווינו ביצועים בין ערימה בינארית, ערימה בינומית וערימת פיבונאצ'י. ערימה בינארית הייתה הכי מהירה גם במקרים גדולים.

```
let bestFirst upperBoundFunc (endTime : DateTime) (prob : KnapsackProblem) :
                                  Solution * DateTime option =
  let priority s = 1.0 / (float <| partialDensity s)
  let startingSolution = Solution.Empty prob
  let mutable bestSolution = startingSolution
  let mutable maybeFindingTime : DateTime option = None
  let solutionsToBranch = new Priority_Queue.BinaryHeap<Solution>()
      solutionsToBranch.Enqueue(startingSolution, priority startingSolution)
  while DateTime.Now < endTime && solutionsToBranch.Count > 0 do
    let sol = solutionsToBranch.Dequeue()
    let openedBits = sol.OpenBits
    let nextIndex = openedBits
    let greedyLeaf = runToLeaf bestSolution.Price upperBoundFunc sol
    if greedyLeaf.Price > bestSolution.Price then
       bestSolution <- greedyLeaf
       if bestSolution.Price = greedyLeaf.Prob.Optimal then
          maybeFindingTime <- Some(DateTime.Now)
    if openedBits < sol.Prob.Items.Length then
      let with1, with0 = sol.Branch
       if with0.IsValid && not <|
          with0.ShouldPrune (bestSolution.Price, upperBoundFunc)
       then solutionsToBranch.Enqueue(with0, priority with0)
       if with1.IsValid && not <|
           with1.ShouldPrune (bestSolution.Price, upperBoundFunc)
       then solutionsToBranch.Enqueue(with1, priority with1)
  (bestSolution, maybeFindingTime)

let rec runToLeaf (bestPrice : int) (upperBoundFunc) (sol : Solution) : Solution =
  if sol.OpenBits = sol.Prob.Items.Length then
     sol
  else
     let with1, with0 = sol.Branch
     [with0; with1]
     |> List.where (fun i -> i.IsValid)
     |> List.where (fun i -> not (i.ShouldPrune(bestPrice, upperBoundFunc)))
     |> List.sortByDescending partialDensity
     |> List.tryHead
     |> fun x -> match x with
                 | None   -> sol
                 | Some a -> runToLeaf bestPrice upperBoundFunc a
```

מיון החפצים:

נמיין את החפצים ע"פ היוריסטיקה, וכך קודם נחפש בתתי העצים של החפצים עם ההיוריסטיקה הטובה יותר, ונקבל גיזומים טובים יותר.

חשבנו רבות על היוריסטיקת מיון. כמובן שיש לקחת בחשבון את יחס המחיר למשקל מבין כל השקים עבור כל מוצר, אך ישנה בעיה של רב מימדיות. כל מוצר משפיע רבות על שאר המוצרים. אם יש הרבה מוצרים שתופסים מקום משק אזי כדאי לקחת יותר מוצרים שתופסים קצת מהשק הזה.

היוריסטיקת המיון שלנו היא

$$\frac{value_{item}}{avg\left\{\frac{weight_{item,sack}}{capacity_{sack}}\right\}}$$

כך, אם הערך של המוצר גבוה יותר, והמשקל הסגולי (הצפיפות) הממוצעת שלו גבוהה יותר, ההיוריסטיקה תהיה גבוהה יותר ונבחר אותו מוקדם יותר במיון ובעץ החיפוש.

```
let runSorted alg upperBoundFunc (problem : KnapsackProblem) =
  let itemHeuristicValue (item : Item) =
  let avgConstraint =
        problem.Knapsacks |> Array.averageBy (fun k ->
                                let cons = double <| item.ConstraintOf k
                                let capacity = double <| k.Capacity
                                cons / capacity)
        (double item.Price) / avgConstraint

  let sortedItems = problem.Items |> Array.sortByDescending (itemHeuristicValue)
  let newProblem = KnapsackProblem(problem.Name, sortedItems,
                                   problem.Knapsacks, problem.Optimal)
  runAlg alg upperBoundFunc newProblem
```

המיון משפר בהרבה מאוד גם את הDFS וגם את הBFS.

ללא המיון האלגוריתם היה מוצא בערך 50% מהפתרון האופטימלי, ועם מיון אנחנו מוצאים 100% מהפתרון האופטימלי בבעיית הSENTO1 בשנייה אחת.

Branch & Bound Analysis

השוואה בין DFS וBFS (המורכב לרוחב)

BFS יותר טוב מDFS רק כאשר אנו לא משתמשים בSorting, ובמקרים אלה, BFS ללא Sorting מגיע לתוצאות טובות מאוד, קצת פחות מאשר DFS/BFS עם Sorting.

השוואה בין Unbounded וFractional Upper Bound:

ישנו שיפור משמעותי מאוד באופן בלתי תלוי בשאר הקונפיגורציות.

בDFS Unsorted, השיפור בא לידי ביטוי בקפיצה מ9 בעיות שמגיעים אליהם לאופטימום אל 24 בעיות שמגיעים אליהם לאופטימום.

כמו כן בDFS Sorted, השיפור בא לידי ביטוי בקפיצה מ26 בעיות שמגיעים אליהם לאופטימום אל 34 בעיות שמגיעים אליהם לאופטימום.

השוואה בין Unsorted וSorted:

ישנו שיפור מאוד משמעותי, באופן בלתי תלוי בשאר הקונפיגורציות.

בDFS Unbounded ישנה קפיצה מ9 ל26, ובDFS Fractional יש קפיצה מ24 ל34.

בBFS Unbounded ישנה קפיצה מ10 ל35 وBFS Fractional ישנה קפיצה מ15 ל35.

סיכום תוצאות הניסויים עבור שניית ריצה אחת:

DFS Unsorted Unbounded:

- 100% מהאופטימלי: 9/55
- 90%< מהאופטימלי: 27/55
- המינימום הוא 55% עבור PET7.

DFS Unsorted Fractional:

- 100% מהאופטימלי: 24/55
- 90%< מהאופטימלי: 42/55
- רק 3 בעיות 80%< מהאופטימלי, מינימום 64%: SENTO1, SENTO2, PET7

BFS Unsorted Unbounded:

- 100% מהאופטימלי: 10/55
- כל השאר מתפלגים אחיד עד ל50%

BFS Unsorted Fractional:

- 100% מהאופטימלי: 15/55
- כל השאר מתפלגים אחיד עד ל50%

הערה:

אם ניתן שנייה וחצי לBFS Unsorted, התוצאות יהיו כמעט כמו של הSorted בDFS וBFS, 34 בעיות עבורם מצאנו פתרון אופטימלי.

DFS Sorted Unbounded:

- 100% מהאופטימלי: 26/55
- כל השאר מעל 95%

DFS Sorted Fractional:

- 100% מהאופטימלי: 34/55
- כל השאר מעל 99%

BFS Sorted Unbounded:

- 100% מהאופטימלי: 35/55
- מעל 98%: 51/55
- כל השאר מעל 90%

BFS Sorted Fractional:

אותו הדבר כמו BFS Sorted Unbounded.

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:

$$1 - \frac{\sum value}{Optimum \sum Value}$$

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 שפיתחנו.

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));
    }
}
```

2. 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];
            }
        }
    }
}
```

3. 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,
                           fitnessMappings = Array())

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

    geneticEngine(geneSimilarity, normalGeneration, localOptimaGeneration)
  }
}
```

שיפור המנוע הגנטי

Survival Selection:

כעת ניתן לבחור כל תת קבוצה של $\{Elitism,\ Random\ Immigrants\}$.

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, וBaldwin 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 $[0, 0.5]$ בהתאם לכמות הזמן שלקח להגיע לפתרון האופטימלי.

אם לא הגענו לפתרון האופטימלי, הFitness יהיה בטווח של $[0.5, 1]$ 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 ה Meta Genetic.
- זיהוי אופטימום לוקאלי ע"י זיהוי חזרת אותו האופטימום בהרבה דורות שיפור מאוד, מאוד משמעותית את אחוזי תוצאות ההתכנסות בMD-Knapsack (45%~ ל88% בPET7), אך מפאת מגבלת הזמן לא נצרף מימוש זה במעבדה 3 וגם בדו"ח. נעשה זאת בהמשך.

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 מ0.5% 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.

שאלה 3 – סיכום תוצאות

תוצאות הניסויים על הבעיות בעזרת האלגוריתם הגנטי

מצאנו את הפתרון האופטימלי לכל הבעיות, אך לא תמיד.

לכן ניתחנו את אחוזי ההצלחה – כמה ריצות מתוך כלל הריצות הצליחו להתכנס לאופטימום.

בנוסף עקבנו אחר התוצאה שהגענו אליה בכישלונות, ומה המרחק שלהם מהפתרון האופטימלי.

ניתן זמן מקסימלי של שנייה אחת לכל בעייה. אך לפי מה שניתן לראות בעמודת הAvg Success Time, נתינת זמן נוסף לא תוביל להגעה לפתרון, כי אנחנו נכנסים למינימום לוקאלי שאנחנו לא מצליחים לצאת ממנו, למרות כל המנגנונים להתמודדות עם מינימום לוקאלי.

הטבלה המלאה עם נתונים נוספים מצורפת לדו"ח זה בפורמט csv.

הטבלה ממויינת לפי אחוזי הצלחה, ולאחר מכן ע"פ זמן ההתכנסות.

Problem name	Success Rate	Avg success time (ms)	Avg Success Iterations	Success Score	Avg Failure score	Avg Failure Percent of best
PET7	0.46	337.49	654.85	16537	16467	99.574
WEING7	0.6	527.33	769.22	1095445	1095354	99.992
WEING8	0.68	120.30	485.19	624319	619366	99.207
SENTO2	0.74	521.47	238.36	8722	8712	99.884
HP2	0.74	176.28	617.39	3186	3117	97.828
PET6	0.81	67.07	164.80	10618	10601	99.844
WEISH23	0.83	327.15	679.88	8344	8341	99.964
PB2	0.89	111.60	391.89	3186	3124	98.048
PB7	0.92	240.29	231.71	1035	1034	99.903
HP1	0.94	32.65	129.95	3418	3404	99.595
SENTO1	0.95	376.84	286.61	7772	7761	99.858
PB1	0.96	26.32	114.04	3090	3076	99.547
WEISH26	0.97	165.29	307.77	9584	9580	99.958
WEISH24	0.98	347.55	496.11	10220	10215	99.951
WEISH25	1	282.66	476.01	9939	9939	100
WEISH30	1	256.46	352.86	11191	11191	100
WEISH18	1	171.55	284.71	9580	9580	100
WEISH29	1	168.16	319.17	9410	9410	100
WEISH28	1	154.01	283.17	9492	9492	100
WEISH27	1	119.08	214.42	9819	9819	100
WEISH21	1	115.88	245.23	9074	9074	100
WEISH20	1	114.05	222.93	9450	9450	100
WEISH22	1	103.01	209.80	8947	8947	100
WEISH17	1	81.03	138.68	8633	8633	100
WEISH14	1	78.22	213.41	6954	6954	100

Problem name	Success Rate	Avg success time (ms)	Avg Success Iterations	Success Score	Avg Failure score	Avg Failure Percent of best
WEISH16	1	69.28	177.91	7289	7289	100
WEISH19	1	63.16	162.60	7698	7698	100
PB6	1	59.51	109.58	776	776	100
WEISH15	1	55.36	152.93	7486	7486	100
WEISH11	1	37.48	140.01	5643	5643	100
WEISH10	1	33.78	107.02	6339	6339	100
PB5	1	33.38	118.80	2139	2139	100
WEISH13	1	30.27	99.91	6159	6159	100
WEISH12	1	29.52	95.88	6339	6339	100
WEISH06	1	24.98	82.44	5557	5557	100
WEISH08	1	23.34	73.72	5605	5605	100
PET5	1	22.80	51.03	12400	12400	100
WEISH07	1	22.63	76.91	5567	5567	100
WEISH09	1	13.76	56.73	5246	5246	100
WEISH02	1	11.17	49.83	4536	4536	100
PET4	1	9.07	29.96	6120	6120	100
WEISH01	1	8.43	38.45	4554	4554	100
WEISH03	1	8.07	39.98	4115	4115	100
WEING6	1	7.77	55.68	130623	130623	100
WEING1	1	7.59	48.59	141278	141278	100
WEING2	1	6.98	50.64	130883	130883	100
WEING4	1	6.92	46.93	119337	119337	100
PB4	1	6.50	40.73	95168	95168	100
WEISH05	1	6.37	33.13	4514	4514	100
WEISH04	1	5.73	30.53	4561	4561	100
WEING5	1	5.69	47.11	98796	98796	100
WEING3	1	4.87	41.97	95677	95677	100
PET3	1	3.25	9.59	4015	4015	100
PET2	1	0.58	1.52	87061	87061	100

ניתוח הזמן והמקום של האלגוריתמים

Branch & Bound

האלגוריתם של Branch and Bound רץ בזמן אקספוננציאלי במקרה הגרוע – כמעט כל ה2^n האפשרויות לבחירות החפצים. כל שיש לנו זו היוריסטיקה שלפעמים עוזרת ולעיתים לא. מבחינת סיבוכיות מקום – באלגוריתם ה-DFS סיבוכיות המקום הינה $O(\#_{items})$ – כעומק עץ מרחב החיפוש, כלומר לינארי עם גודל הבעיה. לעומת זאת, עבור אלגוריתם ה BFS, ניתן להגיע לסיבוכיות מקום אקספוננציאלית, מכיוון שאנחנו מפתחים חזית של פתרונות חלקיים, ובמקרה הגרוע ביותר החזית שלנו תתקדם לפי רמות, וכשנגיע לרמה לפני האחרונה, אנו מחזיקים את כל העלים, ומספר העלים שווה ל2^n. למרות זאת, במבחן המציאות האלגוריתם לא לקח יותר מדי מקום. ה-RAM שהתוכנית לקחה נותר סביר (לכל היותר 200MB כולל המקביליות x4).

אם כן, מבחינת סיבוכיות מקום – ה-DFS קומפקטי במקום בעוד שה-BFS עלול לתפוס כמות מקום אקספוננציאלית.

מבחינת סיבוכיות זמן – כל שיש בידינו זו היוריסטיקה. מדובר בבעיה NP קשה ולכן לא נצליח בבעיה כמו שלנו לפתור אותה היטב **תמיד** באופן דטרמיניסטי. הבעיה תיפתר במהרה עבור מקרים של מספר עצמים קטן, אך עבור מספר עצמים גדול – לא מובטח לנו פתרון טוב.

מדוע הBFS לא תמיד הרבה יותר טוב מDFS?
כאשר מדובר כבר בהיוריסטיקות טובות, הGain שלנו מBFS קטן, אך המחיר שלו בזמן ריצה ובמקום גבוה מאוד ביחס לDFS, ולכן הכדאיות נמוכה יותר.

הגישות האלגוריתמיות של Branch & Bound אינם סקאלאביליים למדי. הם עשויים לקחת זמן ריצה אקספוננציאלי. לצורך שיפור ביצועים באה לטובתינו ההיוריסטיקה של ה Bound וקטימת ערכים שערכם הפוטנציאלי כבר ידוע כלא מספיק – אך עדיין, זמן הריצה עשוי להישאר אקספוננציאלי, ובמקרים גדולים מספיק, הBFS וה-DFS פשוט לא יצליחו להתמודד עם הבעיה.

Genetic MD-Knapsack

סיבוכיות המקום של האלגוריתם קבועה, ותלויה רק בגודל האוכלוסיה שנבחר.

זמן הריצה של האלגוריתם תחרותי מאוד, לכל היותר 550 ms לקבלת פתרון אופטימלי עבור קבוצת הבעיות הנתונה.

הצלחנו לפתור את כל בעיות הדוגמא, שגודלם מגיע גם למעל 100 פריטים.

לכן אנו מאמינים שגישה זו סקלבילית מאוד גם לגדלי בעיות גדולות יותר.

נתוני הניסויים עבור Branch & Bound

התוצאה הטובה ביותר שהאלגוריתם הגיע אליה תוך שנייה אחת

	Percent of Optimum							
Name	**DFS not sorted Bounded**	**DFS not sorted Unbounded**	**DFS sorted Bounded**	**DFS sorted Unbounded**	**BFS sorted Bounded**	**BFS sorted Unbounded**	**BFS not sorted Bounded**	**BFS not sorted Unbounded**
FLEI	100	100	100	100	100	100	100	99.21
HP1	99.53	99.09	99.53	99.53	98.1	98.45	98.51	98.51
HP2	94.41	87.76	99.34	99.09	93.44	94.19	97.96	97.96
PB1	99.48	99	99.48	99.48	98.45	98.45	97.73	97.73
PB2	94.41	91.81	99.34	99.09	93.44	94.19	97.96	97.96
PB4	100	100	100	100	100	100	100	100
PB5	100	100	100	100	100	100	100	100
PB6	98.2	98.2	100	100	90.59	90.59	90.46	90.46
PB7	80.58	80.58	100	100	99.03	99.03	95.17	95.17
PET2	100	100	100	100	100	100	100	100
PET3	100	100	100	100	100	100	100	100
PET4	99.84	99.84	99.84	99.84	99.84	99.84	99.84	99.84
PET5	99.84	93.55	99.84	99.84	99.84	99.84	99.84	99.84
PET6	98.11	75.61	99.48	99.46	98.55	98.55	99.34	99.34
PET7	66.17	55.43	99.57	99.18	98.95	98.95	98.01	98.01
SENTO1	62.39	56.36	100	98.94	98.65	98.65	99.07	99.07
SENTO2	79.63	77.72	100	99.99	99.71	99.71	97.58	97.58
WEING1	100	98.58	100	100	100	100	100	100
WEING2	100	97.35	100	100	100	100	100	100
WEING3	100	95.57	100	100	100	100	100	100
WEING4	100	99.69	100	100	100	100	100	100
WEING5	100	91.18	100	100	100	100	100	100
WEING6	99.7	96.74	99.7	99.7	99.7	99.7	99.7	99.7
WEING7	99.97	99.95	99.97	99.97	99.75	99.75	99.96	99.96
WEING8	91.81	80.92	99.48	99.48	96.83	96.83	90.49	93.43

	Percent of Optimum							
Name	**DFS not sorted Bounded**	**DFS not sorted Unbounded**	**DFS sorted Bounded**	**DFS sorted Unbounded**	**BFS sorted Bounded**	**BFS sorted Unbounded**	**BFS not sorted Bounded**	**BFS not sorted Unbounded**
WEISH01	100	100	100	100	100	100	100	100
WEISH02	99.54	99.54	99.54	99.54	99.54	99.54	99.54	99.54
WEISH03	100	100	100	100	100	100	100	100
WEISH04	100	100	100	100	100	100	100	100
WEISH05	100	100	100	100	100	100	100	100
WEISH06	100	91.4	100	100	100	100	100	100
WEISH07	100	92.99	100	100	100	100	100	100
WEISH08	100	90.88	100	100	100	100	100	100
WEISH09	100	73.39	100	100	100	100	100	100
WEISH10	100	87.14	100	100	100	100	100	100
WEISH11	100	91.62	100	100	100	100	100	100
WEISH12	100	86.07	100	100	100	100	100	100
WEISH13	100	88.24	100	98.85	100	100	100	100
WEISH14	96.61	84.27	100	100	100	100	100	100
WEISH15	99.63	62.73	99.63	99.06	99.63	99.63	99.63	99.63
WEISH16	100	81.7	100	100	100	100	100	100
WEISH17	100	88.01	100	99.9	100	100	100	100
WEISH18	82.6	78.18	99.93	99.53	99.93	99.93	99.5	99.5
WEISH19	93.27	77.1	100	98.39	100	100	97.25	98.44
WEISH20	94.81	76.24	100	99.66	100	100	100	100
WEISH21	96.46	74.13	100	98.5	100	100	100	100
WEISH22	84.56	69.93	99.8	99.56	99.8	99.8	99.68	99.68
WEISH23	89.02	73.32	100	95.41	100	100	99.62	99.62
WEISH24	83.23	76.86	100	98.91	100	100	100	100
WEISH25	96.5	75.33	100	99.71	100	100	100	100
WEISH26	87.46	64.16	100	98.23	100	100	100	100
WEISH27	85.18	65.23	100	100	100	100	100	100
WEISH28	87.97	66.53	100	96.28	100	100	100	100
WEISH29	88.51	66.78	100	96.29	100	100	99.64	99.64
WEISH30	87.94	67.44	99.96	98.61	99.96	99.96	99.96	99.96

זמן התכנסות של האלגוריתם לפתרון, לכל היותר שנייה אחת

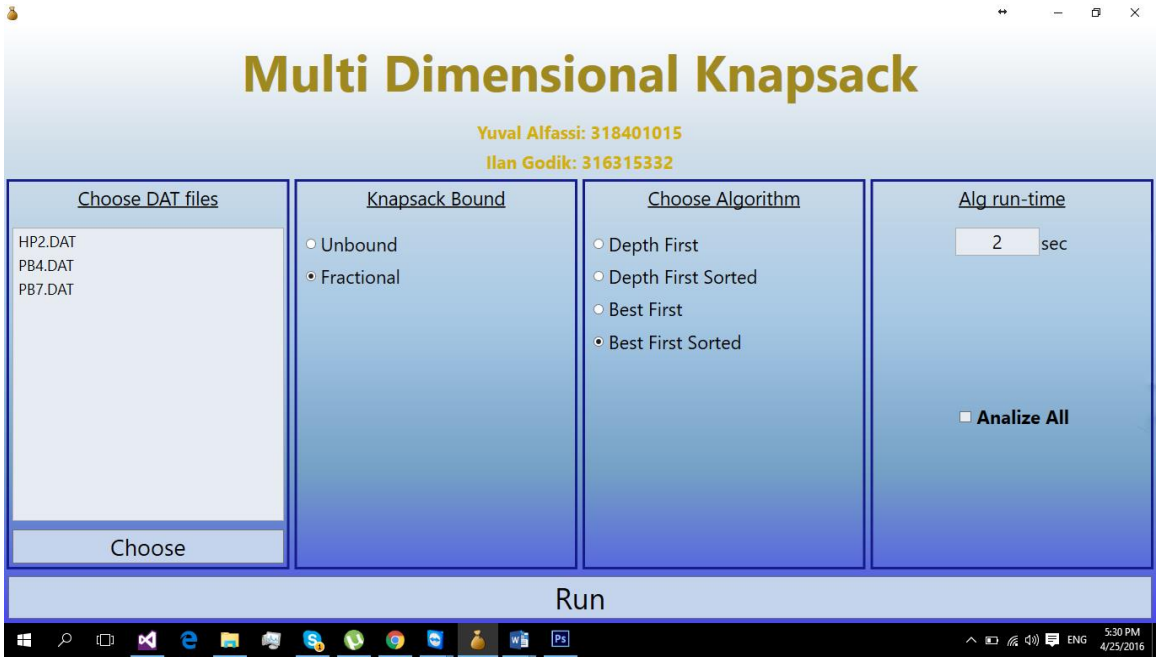
	Percent of Optimum							
Name	**DFS not sorted Bounded**	**DFS not sorted Unbounded**	**DFS sorted Bounded**	**DFS sorted Unbounded**	**BFS sorted Bounded**	**BFS sorted Unbounded**	**BFS not sorted Bounded**	**BFS not sorted Unbounded**
FLEI	0.281	0.417	0.125	0.154	0.353	0.129	0.832	FALSE
HP1	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
HP2	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
PB1	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
PB2	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
PB4	0.004	0.095	0.022	0.103	0.033	0.027	0.304	0.204
PB5	0.141	0.476	0.073	0.113	0.223	0.147	0.917	0.868
PB6	FALSE	FALSE	0.049	0.018	FALSE	FALSE	FALSE	FALSE
PB7	FALSE	FALSE	0.23	0.6	FALSE	FALSE	FALSE	FALSE
PET2	0.001	0.001	0.001	0	0.003	0.002	0.003	0.003
PET3	0.006	0.006	0.001	0.002	0.002	0.001	0.03	0.051
PET4	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
PET5	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
PET6	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
PET7	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
SENTO1	FALSE	FALSE	0.589	FALSE	FALSE	FALSE	FALSE	FALSE
SENTO2	FALSE	FALSE	0.887	FALSE	FALSE	FALSE	FALSE	FALSE
WEING1	0.081	FALSE	0.002	0.007	0.007	0.003	0.001	0.001
WEING2	0.214	FALSE	0.002	0.003	0.001	0.001	0.015	0.012
WEING3	0.051	FALSE	0.001	0.003	0.004	0.003	0.041	0.128
WEING4	0.04	FALSE	0	0	0.008	0.014	0.026	0.037
WEING5	0.043	FALSE	0.001	0.001	0.001	0	0.01	0.01
WEING6	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
WEING7	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
WEING8	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

	Percent of Optimum							
Name	**DFS not sorted Bounded**	**DFS not sorted Unbounded**	**DFS sorted Bounded**	**DFS sorted Unbounded**	**BFS sorted Bounded**	**BFS sorted Unbounded**	**BFS not sorted Bounded**	**BFS not sorted Unbounded**
WEISH01	0.016	0.223	0.001	0.001	0.016	0.008	0.088	0.235
WEISH02	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
WEISH03	0.024	0.132	0.002	0.021	0.053	0.014	0.035	0.028
WEISH04	0.012	0.079	0.001	0.001	0.013	0.009	0.002	0.001
WEISH05	0.014	0.094	0.001	0	0.001	0.001	0.012	0.01
WEISH06	0.186	FALSE	0.009	0.023	0.096	0.071	0.502	0.475
WEISH07	0.146	FALSE	0.002	0.001	0.106	0.033	0.048	0.091
WEISH08	0.135	FALSE	0.002	0.001	0.043	0.025	0.103	0.057
WEISH09	0.058	FALSE	0.002	0.001	0.003	0.002	0.004	0.002
WEISH10	0.416	FALSE	0.006	0.043	0.004	0.003	0.835	0.777
WEISH11	0.334	FALSE	0.007	0.112	0.064	0.027	0.291	0.286
WEISH12	0.658	FALSE	0.003	0.01	0.004	0.002	0.666	0.501
WEISH13	0.456	FALSE	0.007	FALSE	0.053	0.027	0.494	0.282
WEISH14	FALSE	FALSE	0.004	0.002	0.006	0.003	0.406	0.306
WEISH15	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
WEISH16	0.404	FALSE	0.006	0.027	0.114	0.421	0.424	0.304
WEISH17	0.391	FALSE	0.024	FALSE	0.008	0.006	0.637	0.139
WEISH18	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
WEISH19	FALSE	FALSE	0.02	FALSE	0.195	0.121	FALSE	FALSE
WEISH20	FALSE	FALSE	0.013	FALSE	0.008	0.005	0.011	0.008
WEISH21	FALSE	FALSE	0.013	FALSE	0.008	0.005	0.94	0.994
WEISH22	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
WEISH23	FALSE	FALSE	0.078	FALSE	0.277	0.322	FALSE	FALSE
WEISH24	FALSE	FALSE	0.189	FALSE	0.187	0.216	0.495	0.5
WEISH25	FALSE	FALSE	0.043	FALSE	0.22	0.182	0.447	0.523
WEISH26	FALSE	FALSE	0.069	FALSE	0.011	0.008	0.684	0.649
WEISH27	FALSE	FALSE	0.011	0.051	0.275	0.18	0.345	0.367
WEISH28	FALSE	FALSE	0.053	FALSE	0.023	0.008	0.018	0.013
WEISH29	FALSE	FALSE	0.212	FALSE	0.014	0.008	FALSE	FALSE
WEISH30	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

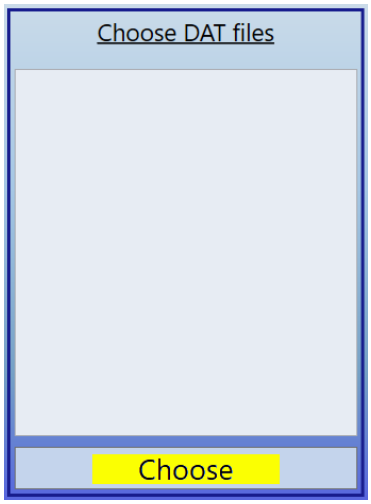
User Interface for Branch & Bound והוראות הרצה

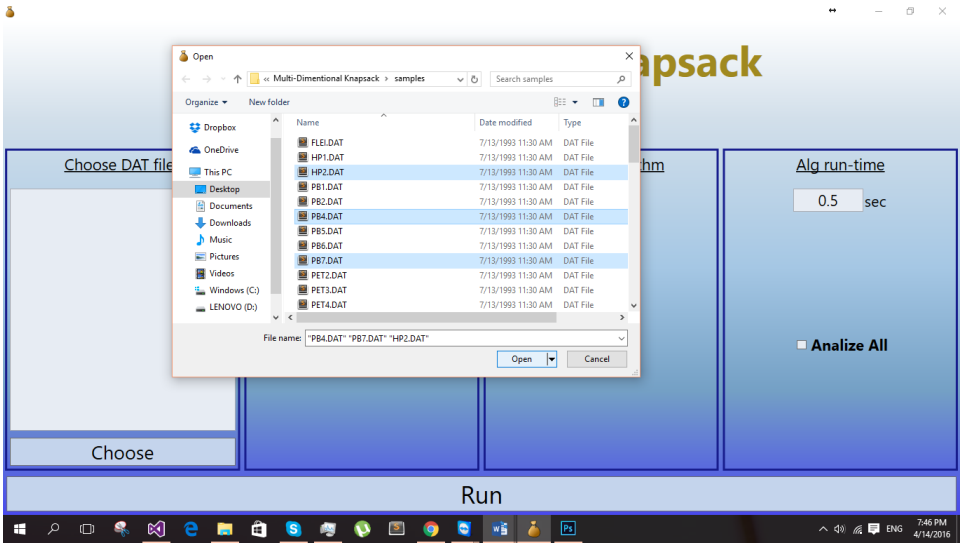
מיקום תכנית ההרצה:

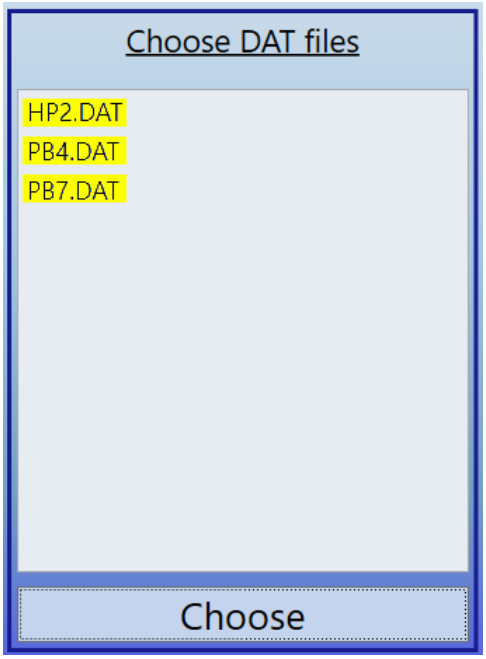
Name	Date modified	Type	Size
Arction.DirectX.dll	4/12/2016 1:51 AM	Application extens...	566 KB
FSharp.Charting.dll	4/12/2016 3:40 PM	Application extens...	485 KB
FSharp.Charting.pdb	4/12/2016 3:40 PM	Program Debug D...	490 KB
FSharp.Charting.xml	4/12/2016 3:40 PM	XML Document	180 KB
FSharp.Core.dll	6/19/2015 6:07 PM	Application extens...	1,471 KB
FSharp.Core.xml	6/19/2015 5:50 PM	XML Document	692 KB
GuiMDKnapsack.exe	4/14/2016 3:42 PM	Application	742 KB
GuiMDKnapsack.exe.config	4/14/2016 2:07 AM	XML Configuratio...	1 KB
GuiMDKnapsack.pdb	4/14/2016 3:42 PM	Program Debug D...	34 KB
GuiMDKnapsack.vshost.exe	4/14/2016 6:30 PM	Application	23 KB
GuiMDKnapsack.vshost.exe.config	4/14/2016 2:07 AM	XML Configuratio...	1 KB
GuiMDKnapsack.vshost.exe.manifest	10/30/2015 9:19 AM	MANIFEST File	1 KB
MDKnapsack.exe	4/14/2016 3:42 PM	Application	53 KB
MDKnapsack.pdb	4/14/2016 3:42 PM	Program Debug D...	76 KB
MDKnapsack.xml	4/14/2016 3:42 PM	XML Document	1 KB
Priority Queue.dll	4/12/2016 2:25 PM	Application extens...	12 KB
Priority Queue.pdb	4/12/2016 2:25 PM	Program Debug D...	28 KB



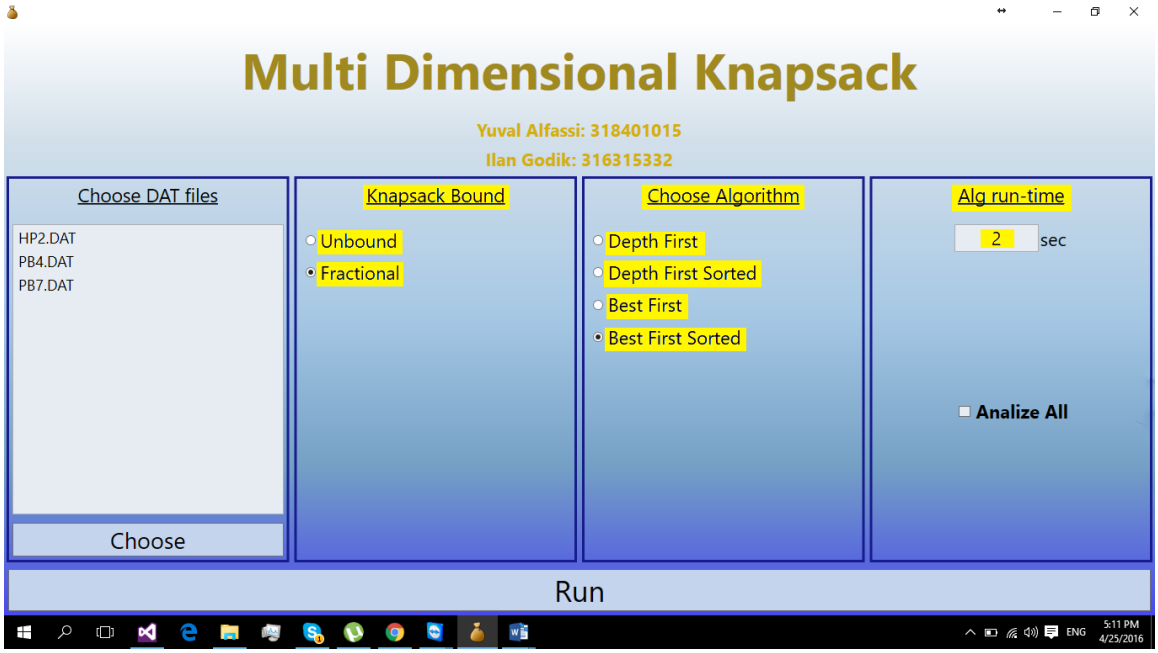
תחילה, יש לבחור קובץ/קבצי DAT להרצה ואנליזה המכילים בעיית multiple dimensional knapsack בפורמט שניתן:







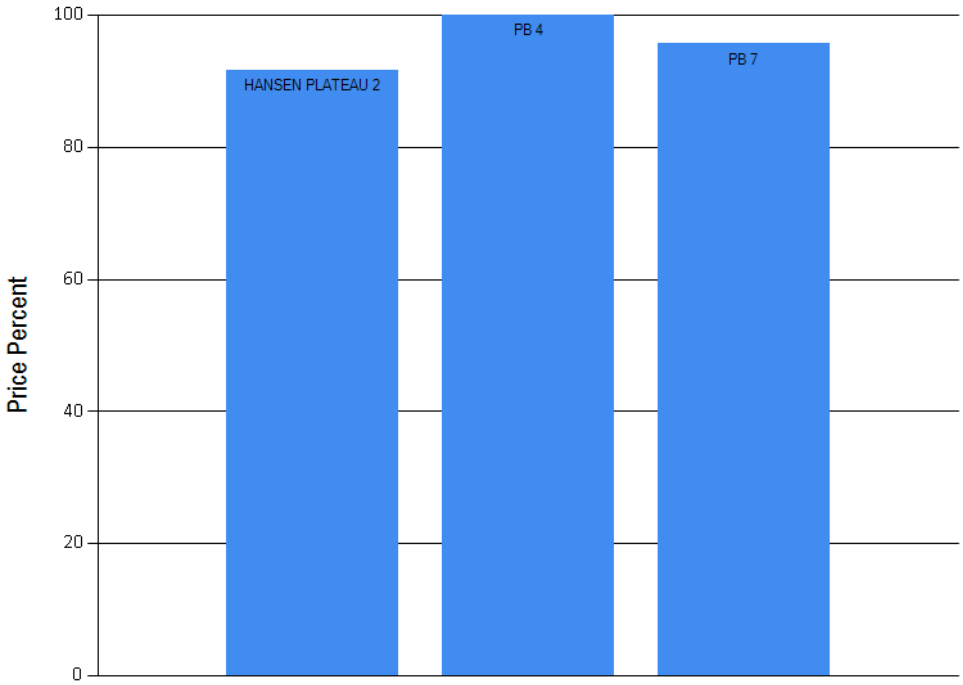
לאחר מכן, יש לבחור פרמטרים להרצת הבעיה: האם לבחור היוריסטיקה של חסם על השק או היוריסטיקה של שק לא מוגבל, האם יש לעשות BFS, DFS והאם יש למיין, וכמו כן, כמה זמן מקצים לריצת כל אלגוריתם.



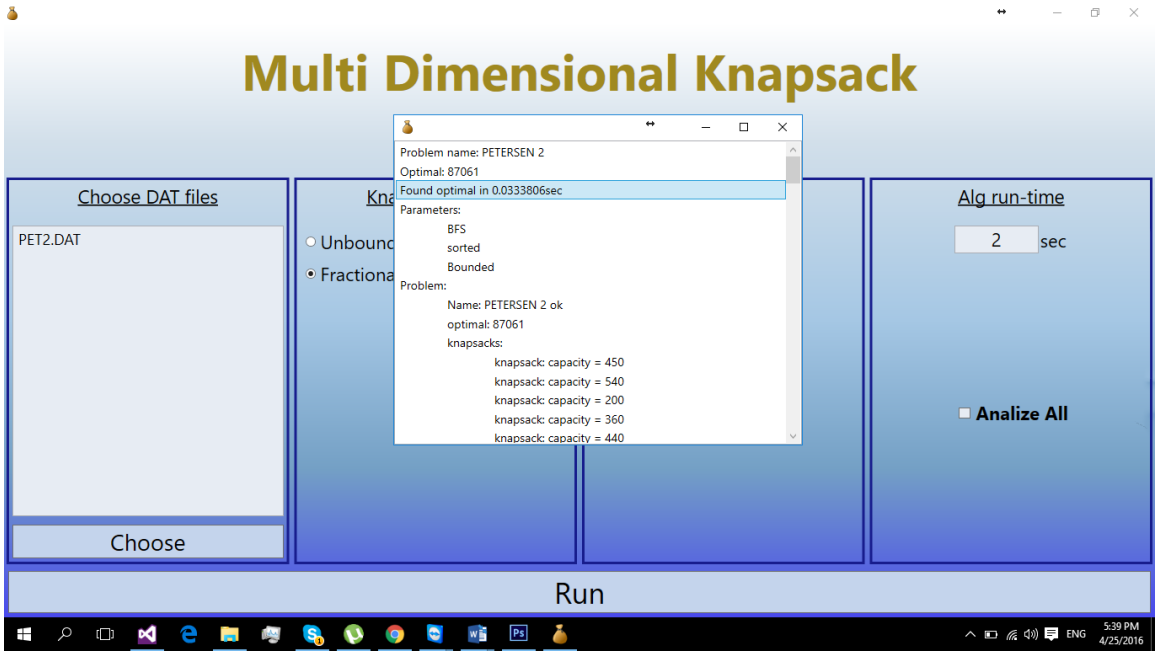
ואז להריץ:

התוצאה: השוואה בין הבעיות מבחינת לאיזה אחוז מהפתרון האופטימאלי הם הצליחו להגיע.

בדוגמא – עבור PB4 האלגוריתם הספיק להגיע בזמן שהוקצב לו לבערך 95% מהאופטימאלי, בעוד שעבור PB7 הגענו רק ל-80% מהאופטימאלי.



אם נריץ רק עבור קובץ DAT אחד, ניראה את המידע על הבעיה, מה הפיתרון המוצע שהגענו אליו – איזה מוצרים לקחת ואיזה לא:



בנוסף, על ידי לחיצה על Analize all, תעשה אנליזה על קובץ ה-DAT הראשון שנבחר לפי כל קומבינציה אפשרית של פרמטרים לבעיה: