מעבדה 3 בבינה מלאכותית

נושא:

רלקסציה של תכנות לינארי

­­

מגישים:

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

מנחה:

מר שי בושינסקי



תאריך הגשה:

15 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

סדר גודל העבודה:

* מעבדה 1: 45 שעות
* מעבדה 2: 50 שעות
* 3750 שורות קוד
* ~150 commits

**מבוא**

**תיאור הבעיה**

עלינו לפתור את בעיית השק הרב מימדית.

נתונים מספר שקים ומספר מוצרים. לכל שק ישנה קיבולת ולכל מוצר יש את מחירו ואת כמה שהוא לוקח מכל שק. עבור כל עצם עלינו להחליט האם ניקח אותו או לאו. אם ניקח אותו – הוא יתפוס מקום בכל השקים. עלינו למקסם את סכום המחירים של המוצרים שלקחנו כאשר נישאר בטווח של משקל השק.

פיתרון טריוויאלי לבעיה – ניסיון כל האופציות האפשריות לבחירה. סיבוכיות זמן – אקספוננציאלית.

מדובר בבעייה NP קשה. בדו"ח זה נסרוק מספר היוריסטיקות איתן ניגשנו לפתור את הבעיה.

**תכנות הבעיה**

את המנוע הגנטי מהשיעורים הקודמים תכנתנו ב scala וב-java. את הפתרון הגנטי לבעיה הרצנו על המנוע הגנטי שכבר פיתחנו במטלה 2.

לצורך התכנות הלינארי, החלטנו להתרענן מעט, ולעבור לתכנת את מטלה 3 בשפות f# ו- c# ממשפחת .net. f# הינה שפה פונקציונאלית, נוחה ורובסטית לשימוש. ב-f# תכנתנו את הלוגיקה והפתרון של הבעיה. עם זאת, פיתחנו GUI נוח אותו רשמנו ב c#.

תחילה פתרנו את הבעיה על ידי קוד פונקציונאלי ו- High level למדי. הרצנו כלי של performance profiling על התוכנית שלנו איתו ראינו היכן יש Bottle-Neck מבחינת מימוש. החלפנו הרבה מהפונקציות בלולאות, השתמשנו ב Bit Vector בתיאור של פיתרון של בעיה – דבר ששיפר רבות את זמן הריצה.

**ארכיטקטורה**

שק – Knapsack, יש לו קיבולת:

type Knapsack(capacity : int)

מוצר – Item, יש לו מחיר ומשקל עבור כל שק

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

תיאור בעיה – יש שם לבעיה, ישנם שקים, מוצרים, ונתון לנו מראש מה הערך האופטימאלי אליו עלינו לשאוף

type KnapsackProblem(name : string, items : Item array,

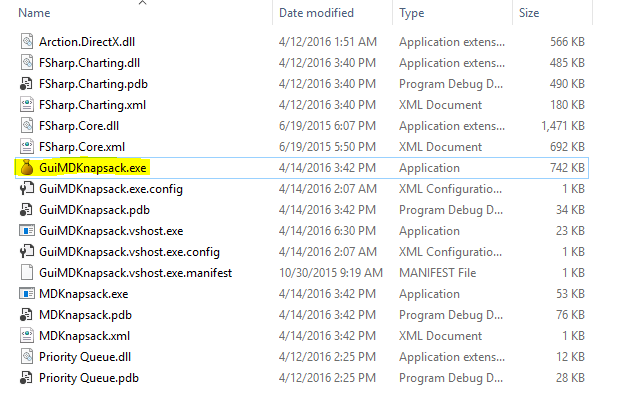
knapsacks : Knapsack array, optimal : int)

תיאור פתרון בעיה – יש לנו את המוצרים איתם הבעיה מתמודדת, ו- Bit Vector המציין עבור כל מוצר האם לוקחים אותו

type Solution(itemsTaken : BitArray, items : Item array)

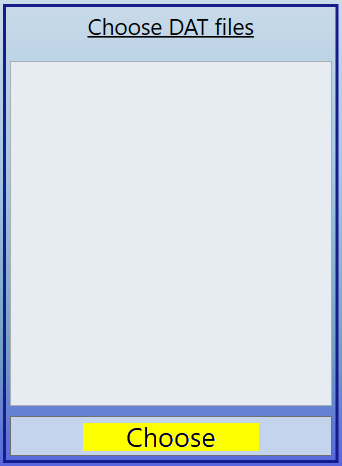
**User Interface**

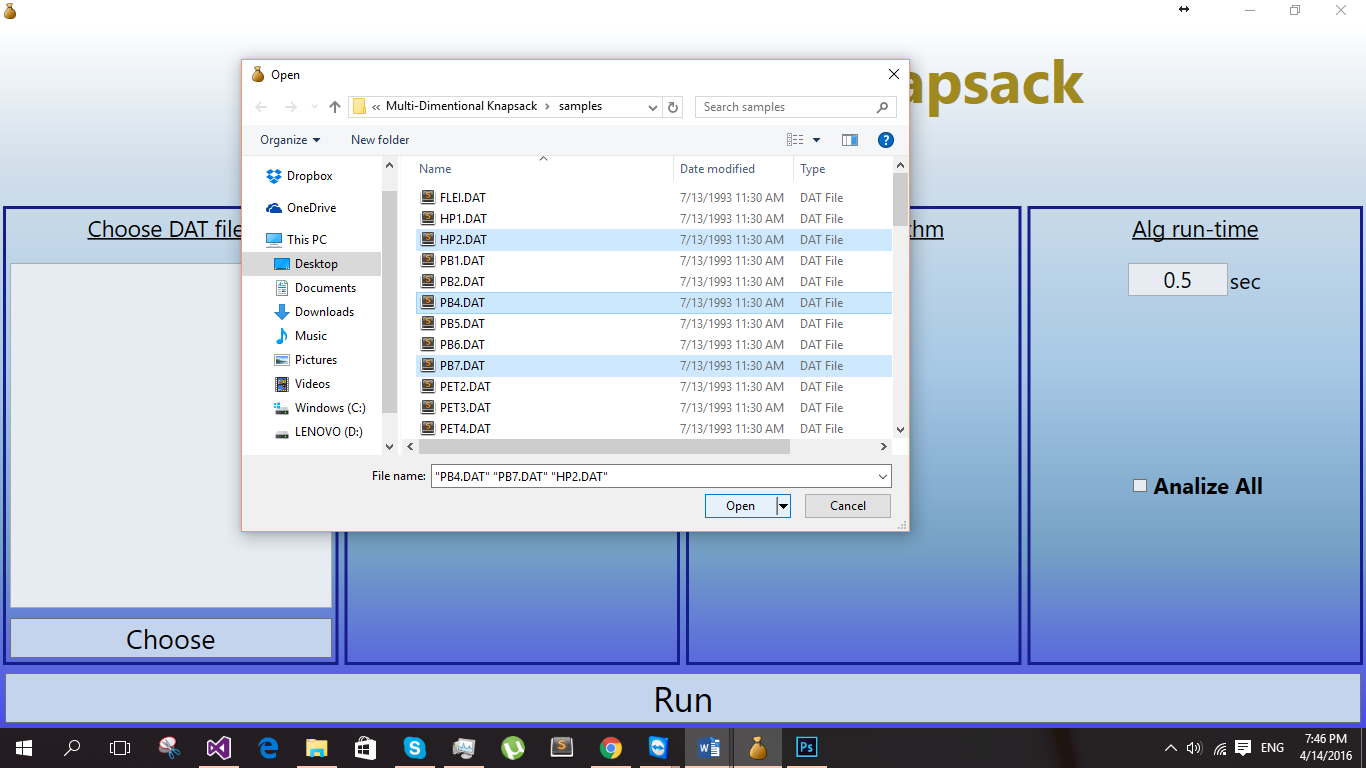
התוכנה:

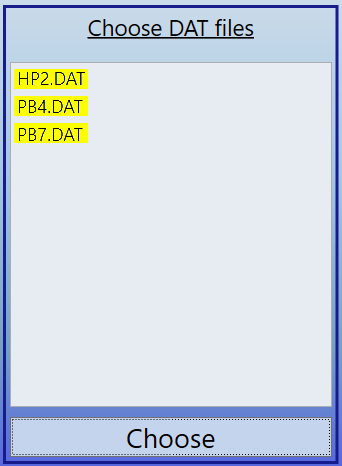




תחילה, יש לבחור קובץ/קבצי DAT להרצה ואנליזה המכילים בעיית multiple dimensional knapsack בפורמט שניתן:







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.



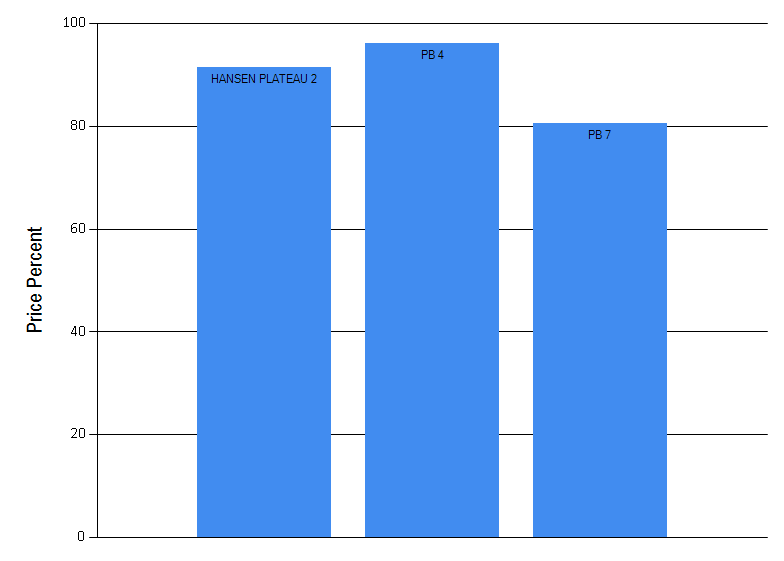
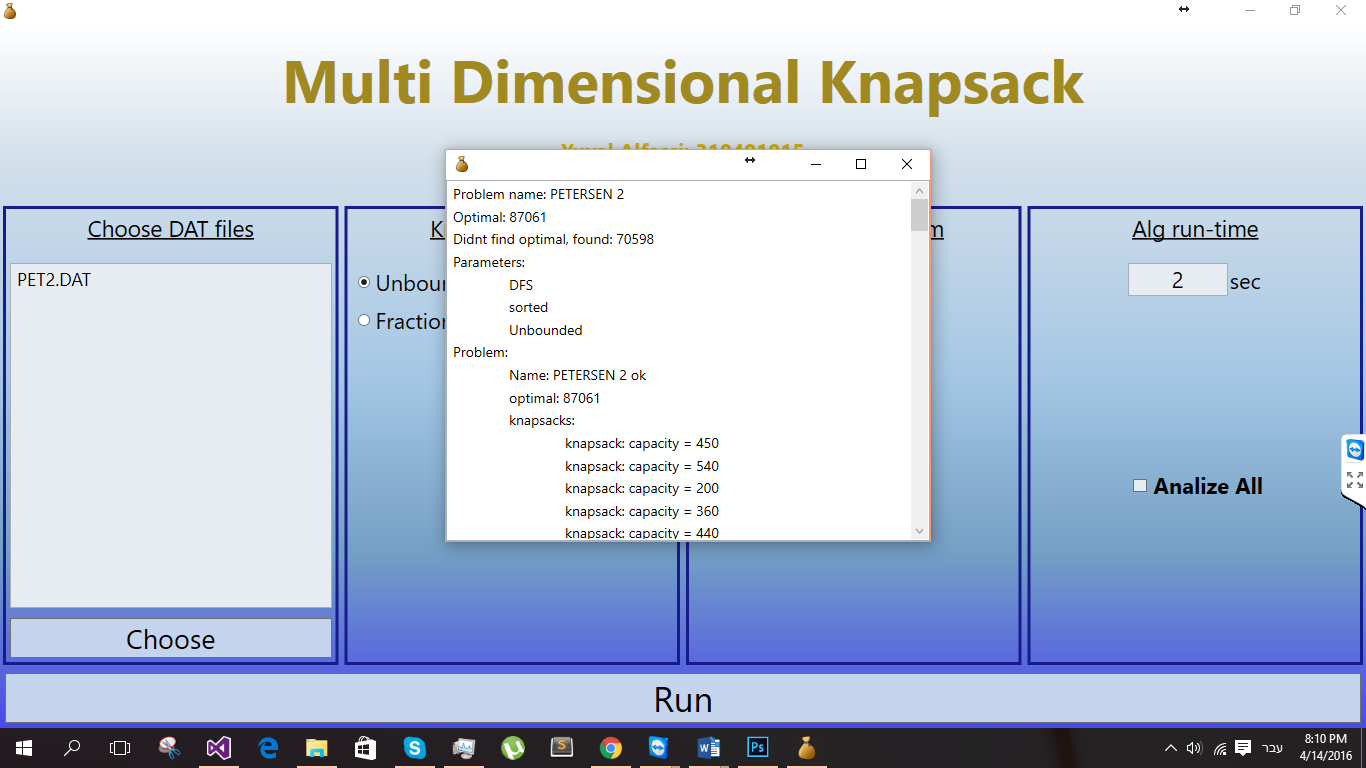
ואז להריץ:



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

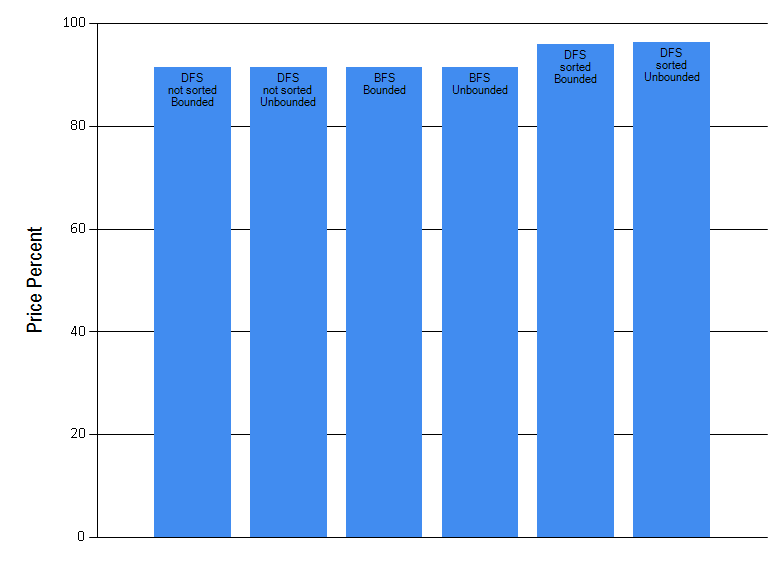
בדוגמא – עבור PB4 האלגוריתם הספיק להגיע בזמן שהוקצב לו לבערך 95% מהאופטימאלי, בעוד שעבור PB7 הגענו רק ל-80% מהאופטימאלי.

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:





**שאלה 1 – אלגוריתם**

אלגוריתם גנטי.......... אילן עושה

**שאלה 2 – Branch and Bound**

את הבעיה מידלנו על ידי סגנון של Branch and Bound כפי שהוצג בכיתה.

התחלנו ממצב שלא לקחנו שום מוצר. לקחנו את שני 'בניו' – באחד לוקחים מוצר ובאחר לא לוקחים מוצר. אם הערך הפוטנציאלי שלנו בשלב מסויים קטן מהערך שמצאנו – נקטום עץ זה.

שק לא חסום. זה היה למעשה מעין DFS על העץ בוא העדפנו קודם לפתח את הצד בוא כן בחרנו מוצר. קטמנו תתי עצים ברגע שראינו שבחירתינו עד עתה חורגת מקיבולות השקים, או כאשר ראינו שהערך הפוטנציאלי לא יכול אפילו להגיע לערך הטוב ביותר שאספנו עד כה – קטמנו את העץ.

קוד: (סליחה על הלולאות והכל.. זה לצורך האצת ביצועים של פונקציות שרצות הרבה)

בדיקה האם הפתרון אינו חורג מקיבולות השקים:

member x.IsValid (knapsacks : Knapsack array) =

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)

פונקציית ה pruning שלנו – תמיד תחזיק קבוע ערך גבוה מאוד (unbound knapsack):

let unboundedKnapsack : PrunningFunc =

fun \_ \_ \_ \_ -> Int32.MaxValue / 2

קטימה של עצים שערכם הפוטנציאלי קטן מהערך שנמצא עד כה (פונקציית ה- pruning שתועבר תהיה ה- unbounded knapsack):

אם הערך הנוכחי שלנו ועוד הסכום הפוטנציאלי קטן מהמחיר שמצאנו עד כה אז נגיד שכדאי לעשות prune לתת העץ.

member x.ShouldPrune (doneUntilIndex : int, bestPrice : int, prunning : PrunningFunc, knapsacks : Knapsack[]) : bool =

let mutable restPotentialPrice = 0

for i = doneUntilIndex + 1 to itemsTaken.Length - 1 do

restPotentialPrice <- restPotentialPrice + items.[i].Price

let maxPossible = price + restPotentialPrice

maxPossible < bestPrice || prunning knapsacks price items doneUntilIndex < bestPrice2)

שק לא חסום:

עלינו למצוא חסם עליון לפתרון הבעיה, כך שבהינתן פיתרון חלקי, נקבע דטרמיניסטית מחיר המקסימאלי של השק.

עבור כל שק בנפרד, מילאנו אותו בכמות המקסימאלית של היחס מחיר-למשקל, כאשר הנחנו שנוכל ללקחת חלקי מוצר. כך, עבור כל שק נתון לנו כמה מקסימום מחיר יכול להיות עבורו. מבין כל המקסימות הללו לקחנו את המינימום – והוא יהיה חסם עליון למחיר.

אפוא, כל שק ניתן למלא עד למחיר מקסימאלי כלשהו, כל מחיר כזה מהווה חסם עליון. ניקח חסם עליון מינימאלי.

קוד:

let upperBoundFractional (knapsacks : Knapsack[]) (currentPrice : int) (items : Item[]) (takenUntil : int) : int =

knapsacks

|> Seq.ofArray

|> Seq.map (fun k -> fractionedFilledKnapsack items k takenUntil)

|> Seq.map (fun optimisticPrice -> optimisticPrice + currentPrice)

|> Seq.min

2.2.1)

Dfs – חיפוש לעומק. ישנה עדיפות להתקדם קודם לכיוון ה"לקחת מוצר" מאשר לכיוון ה"לא לקחת מוצר". לא נפתח צמתים שערכם הפוטנציאלי נמוך מהערך האופטימי

קוד:

(לולאות וקוד לא קריא --- בשביל שיפור ביצועים)

let dfs (prunning : PrunningFunc) (endTime : DateTime) (items : Item array) (knapsacks : Knapsack array) (knownOpt : int) : Solution \* DateTime option =

let startingSolution = Solution.Empty items

let mutable bestSolution = startingSolution

let mutable maybeFindingTime : DateTime option = None

let solutionsToBranch = new Stack<Solution \* int>()

solutionsToBranch.Push(startingSolution, 0)

while DateTime.Now < endTime && solutionsToBranch.Count > 0 do

let sol, index = solutionsToBranch.Pop()

let nextIndex = index + 1

if nextIndex = items.Length then

if sol.Price > bestSolution.Price then

bestSolution <- sol

if bestSolution.Price = knownOpt then maybeFindingTime <- Some(DateTime.Now)

else

let with1, with0 = sol.Branch nextIndex

if with0.IsValid knapsacks && not <| with0.ShouldPrune(nextIndex, bestSolution.Price, prunning, knapsacks) then

solutionsToBranch.Push(with0, nextIndex)

if with1.IsValid knapsacks && not <| with1.ShouldPrune(nextIndex, bestSolution.Price, prunning, knapsacks) then

solutionsToBranch.Push(with1, nextIndex)

()

(bestSolution, maybeFindingTime)

2.2)

Bfs – חיפוש best first search. נתחזק תור עדיפויות ממנו נפתח בכל שלב את הצומת בעל המחיר הטוב ביותר באופן חמדני. גם כאן נגזום צמתים שערכם נמוך מהערך האופטימי. השתמשנו במחיר של הצומת כערך היוריסטי לפתיחת הצמתים.

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

קוד:

(לולאות וקוד לא קריא --- בשביל שיפור ביצועים. דומה ביותר ל- dfs רק שכאן הפתיחה היא עם תור עדיפויות)

let bestFirst (prunning : PrunningFunc) (endTime : DateTime) (items : Item array) (knapsacks : Knapsack array) (knownOpt : int) : Solution \* DateTime option =

let startingSolution = Solution.Empty items

let mutable bestSolution = startingSolution

let mutable maybeFindingTime : DateTime option = None

let solutionsToBranch = new Priority\_Queue.BinaryHeap<Solution \* int>()

solutionsToBranch.Enqueue((startingSolution, 0), 1.0 / float startingSolution.Price)

while DateTime.Now < endTime && solutionsToBranch.Count > 0 do

let sol, index = solutionsToBranch.Dequeue()

let nextIndex = index + 1

if nextIndex = items.Length then

if sol.Price > bestSolution.Price then

bestSolution <- sol

if bestSolution.Price = knownOpt then maybeFindingTime <- Some(DateTime.Now)

else

let with1, with0 = sol.Branch nextIndex

if with0.IsValid knapsacks && not <| with0.ShouldPrune(nextIndex, bestSolution.Price, prunning, knapsacks) then

solutionsToBranch.Enqueue((with0, nextIndex), 1.0 / float with0.Price)

if with1.IsValid knapsacks && not <| with1.ShouldPrune(nextIndex, bestSolution.Price, prunning, knapsacks) then

solutionsToBranch.Enqueue((with1, nextIndex), 1.0 / float with1.Price)

()

(bestSolution, maybeFindingTime)מיון הרכיבים:

ניתן לשלב dfs יחד עם מיון הרכיבים (dfs לא מתחשב בסדר שלהם). אם נמיין את הפריטים לפי סדר מסויים, ה-dfs יתחיל מאותם פריטים שהיוריסטית יותר כדאי לבחור.

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 (מאוד) על שאר המוצרים. אם יש הרבה מוצרים שתופסים מקום משק אזי כדאי לקחת יותר מוצרים שתופסים קצת מהשק הזה, אך גם לכך יש דוגמא נגדית. כלומר, קשה להתייחס למוצרים מבחינה לוקאלית כיוון שהכל כל כך גלובאלי בגלל שישנה הרבה שקים. לאחר דיונים רבים, החלטנו ללכת על שיטת מיון שתקח בחשבון עבור כל מוצר את המחיר שלו, את כמה שהוא לוקח מכל שק, ואת הגודל של השק. ניקח מכל מוצר את המחיר שלו כפול ממוצע היחס בין כמה שהוא לוקח מכל שק לקיבולת השק. כך, אם מוצר תופס מעט מקום נרצה יותר לקחת אותו, ואם מוצר שווה יותר גם כן נרצה יותר לקחת אותו.

קוד:

let dfsSorted (prunning : PrunningFunc) (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

1.0 - cons / capacity)

(double item.Price) \* avgConstraint

let sortedItems = problem.Items |> Array.sortByDescending (itemHeuristicValue)

let newProblem = KnapsackProblem(problem.Name, sortedItems, problem.Knapsacks, problem.Optimal)

runAlg dfs prunning newProblem

לאחר הרצות רבות, ראינו שלעיתים קרובות המיון עוזר, אך לא תמיד. לעיתים המיון לא משפר. הסבר לכך – זוהי היוריסטיקה. זוהי בעיה NP קשה... תמיד יהיו דברים שלא ייפעלו.

**שאלה 3 – סיכום תוצאות**

אלגוריתם גנטי.......... לחכות לאילן

3.1)

צריך לשלב את הדוחות.

3.2)

האלגוריתם של branch and bound טוב במקרים קטנים. מבחינת סיבוכיות הזמן – סיבוכיות הזמן הינה אקספוננציאלית. כל שיש לנו זו היוריסטיקה שלפעמים פועלת ולעיתים לא. מבחינת סיבוכיות מקום – באלגוריתם ה-dfs סיבוכיות המקום הינה – כעומק עץ ה dfs. לעומת זאת, עבור אלגוריתם ה bfs, ניתן להגיע לסיבוכיות מקום אקספוננציאלית. מפתחים באופן חמדני את הענפים בעלי הערך הטוב ביותר. מתחזקים תור עדיפויות עבור כל הענפים הפעילים. ב-worst case אפשר להגיע לסיבוכיות מקום אקספוננציאלית ב-best first. לעומת זאת, במבחן התוצאה האלגוריתם לא לקח הרבה מקום. ה-RAM שהתוכנית לקחה נותר קטן, הענפים הטובים נחקרו עד תום במהירות, ואף אחד לא פותח במקביל אליהם.

אם כן, מבחינת סיבוכיות זמן – כל שיש בידינו זו היוריסטיקה. מדובר בבעיה NP קשה ולכן לא נצליח בבעיה כמו שלנו לפתור אותה דטרמיניסטית. הבעיה תיפתר במהרה עבור מקרים של מספר עצמים קטן, אך עבור מספר עצמים גדול – לא מובטח לנו פתרון טוב. מבחינת סיבוכיות מקום – ה-dfs קומפקטי במקום בעוד שה-bfs עלול לתפוס כמות מקום אקספוננציאלית.

בנוסף לכך, dfs לעיתים קרובות לא הצליח להתמודד עם הבעיה בהצלחה. הוא חקרו תת עץ מאוד ספציפי והתחלתי ופשוט לא הספיק להתקדם הלאה לתתי עצים אחרים.

לגבי ה bfs והמיון – מה שיש בידינו הינה היוריסטיקה, שלפעמים פועלת ולעיתים לא. לפיכך, לעיתים קרובות לא הצלחנו להגיע לפתרון טוב.

3.3)

האלגוריתמים הללו 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 ה bfs וה-dfs פשוט לא יצליחו להתמודד עם הבעיה.