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.

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

**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 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,

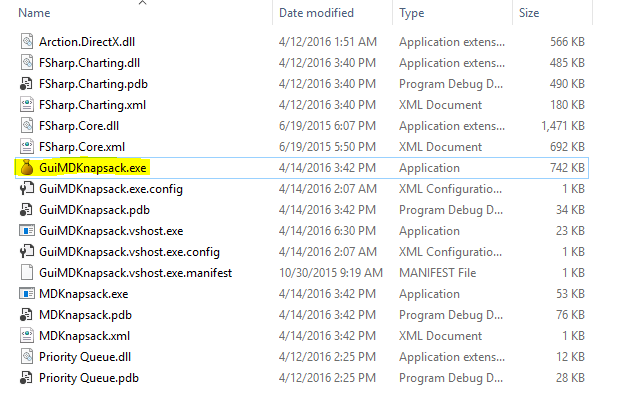
knapsacks : Knapsack array, optimal : int)

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)

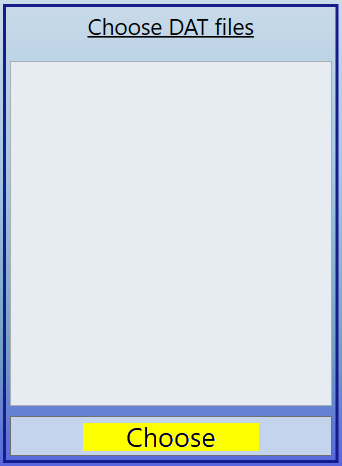
**User Interface**

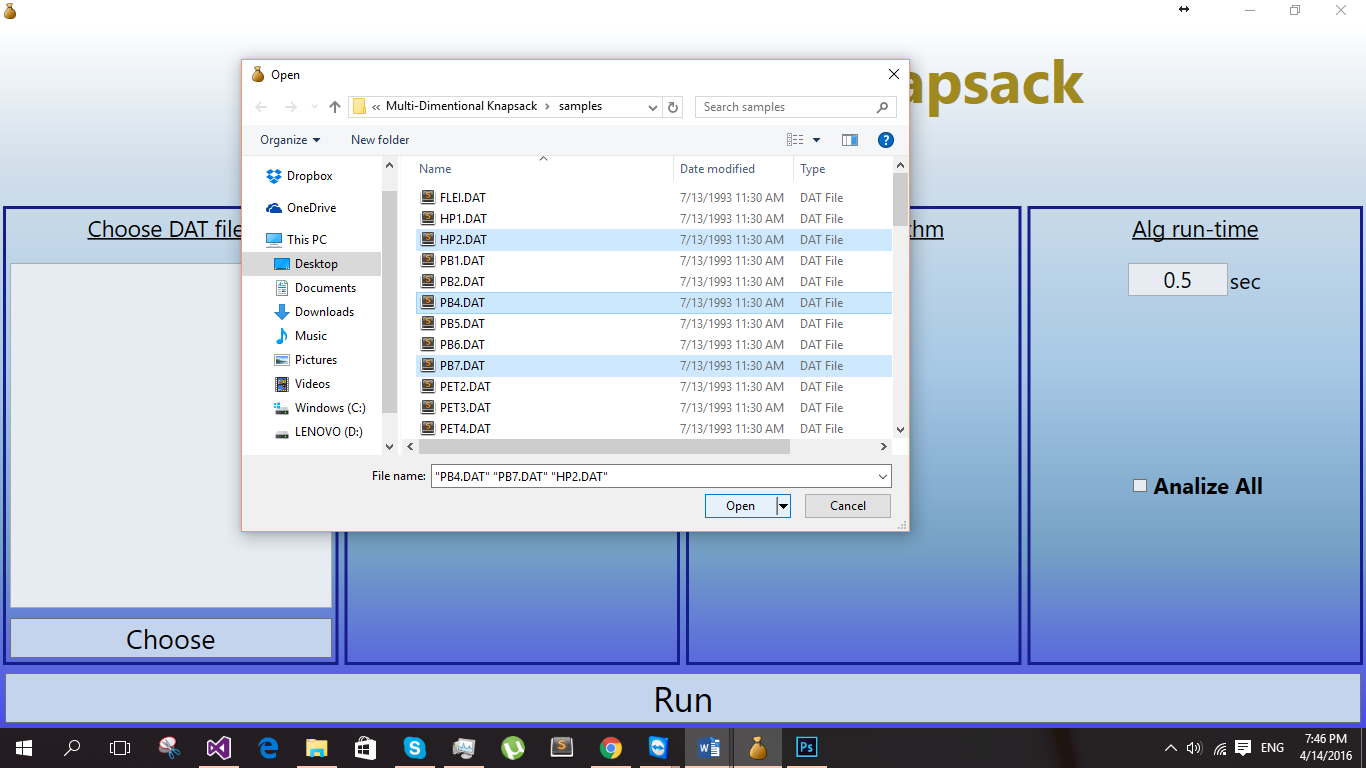
התוכנה:

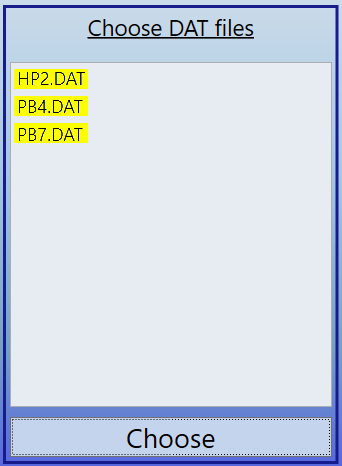




תחילה, יש לבחור קובץ/קבצי DAT להרצה ואנליזה המכילים בעיית multiple dimensional knapsack בפורמט שניתן:







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.



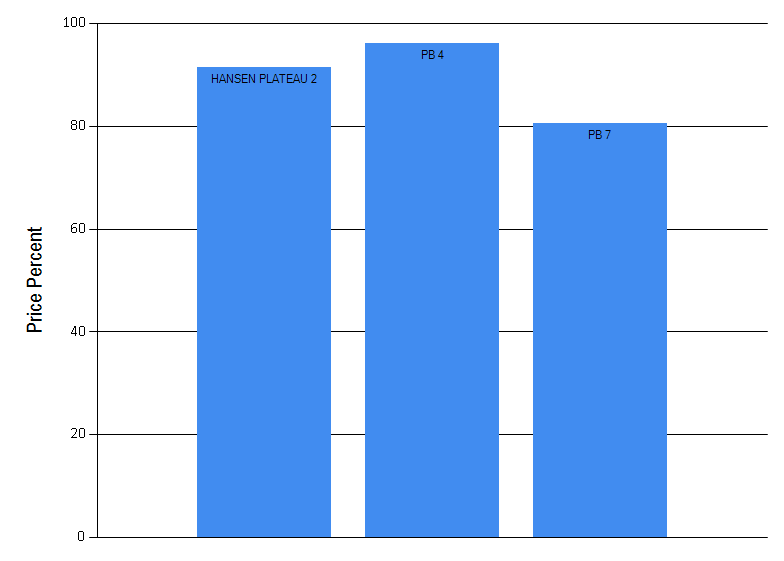
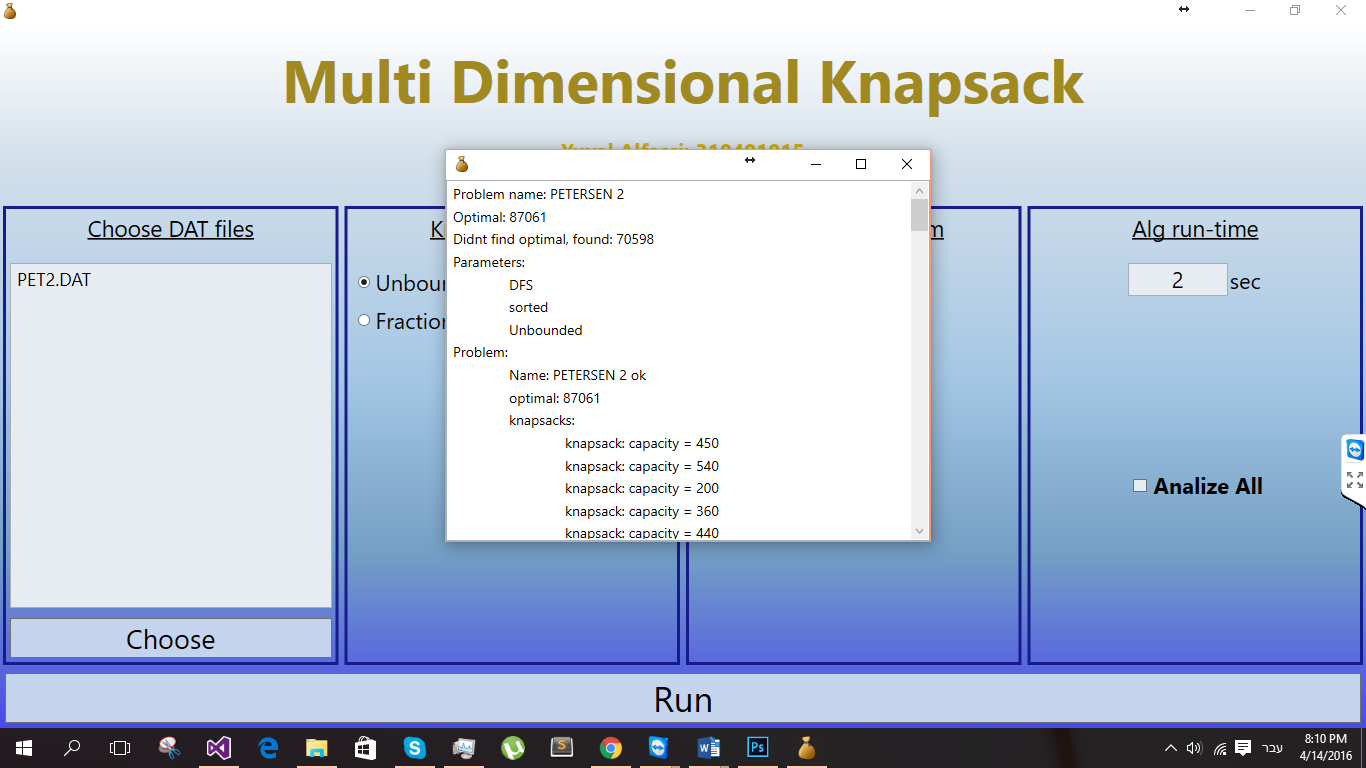
ואז להריץ:



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

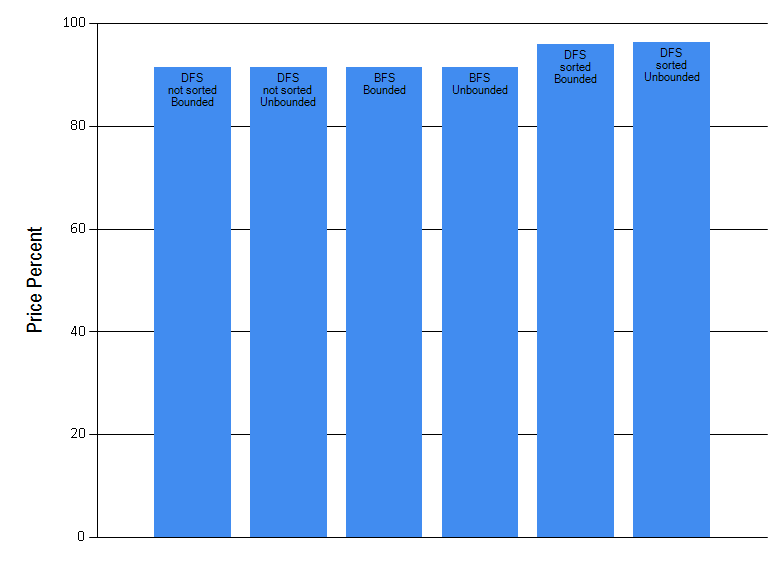
בדוגמא – עבור PB4 האלגוריתם הספיק להגיע בזמן שהוקצב לו לבערך 95% מהאופטימאלי, בעוד שעבור PB7 הגענו רק ל-80% מהאופטימאלי.

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**

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

**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 ה – החלטנו עד כה אילו מ 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

1. **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:

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:

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)

1. **BFS** - חיפוש Best First Search.

בכל בחירה – האם לקחת או לא לקחת את החפץ הנוכחי, נבחר את הילד בו החסם העליון ע"פ הFractional Upper Bound יותר גבוה – כך אנחנו קודם נחפש בתתי עצים יותר מבטיחים, נקבל פתרונות סבירים מהר יותר, ונעשה יותר Cutoffs מוקדם יותר.

let bestFirst 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

[with0; with1]

|> List.where (fun i -> i.IsValid)

|> List.where (fun i ->

not (i.ShouldPrune(bestSolution.Price, upperBoundFuns)))

|> List.sortBy (fun i -> i.PartialDensity)

|> List.iter (fun i -> solutionsToBranch.Push(i))

(bestSolution, maybeFindingTime)

לפי ניסויים, תוצאות של Best First דומות מאוד לתוצאות של DFS.

לכן, לא חשפנו את מימוש זה למשתמש.

1. חיפוש לרוחב מונחה היוריסטיקה

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.

השתמשנו בערימה בינארית לצורך תור העדיפויות. השווינו ביצועים בין ערימה בינארית, ערימה בינומית וערימת פיבונאצ'י. ערימה בינארית היתה 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.SimplePriorityQueue<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

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.Enqueue(with0, priority with0)

if with1.IsValid && not <|

with1.ShouldPrune (bestSolution.Price, upperBoundFunc)

then solutionsToBranch.Enqueue(with1, priority with1)

(bestSolution, maybeFindingTime)

שיטה זו נתנה יתרון מאוד משמעותי על גבי DFS פשוט וBFS.

זהו המימוש שנחשף למשתמש בתור BFS.

מיון הרכיבים:

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

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 40% מהפתרון האופטימלי, ועם מיון אנחנו מוצאים 99% מהפתרון האופטימלי בבעיית הPET7.

**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.