הנחיות לפתרון תרגילי הבית

- על הקוד המוגש להיות מתועד היטב ועליו לכלול:
  - מפרט, כפי שהודגם בתרגול.
  - תיעוד של כל מחלקה ומתודה ושל קטעי קוד רלוונטיים.
  - במידת הצורך, יש להוסיף תיעוד חיצוני.
- יש להפעיל את הכלי Javadoc כדי ליצור קבצי תיעוד בפורמט HTML ולצרף אותם לפתרון הממוחשב המוגש. כדי לגרום לקובצי ה-HTML להכיל את פסקאות המפרט שבהן אנו משתמשים, יש לציין זאת במפורש. ב-Eclipse, ניתן לבצע פעולה זו באופן הבא: 1. לבחור Export מתפריט File, לבחור Java->Javadoc וללחוץ על כפתור Next, 2. לבחור עבור Javadoc command את הקובץ javadoc.exe מתוך התיקייה bin הנמצאת בתיקייה שבה מותקן ה-Java SDK, 3. לבחור את הקבצים שלהם מעוניינים ליצור תיעוד וללחוץ פעמיים על כפתור Next, 4. להקיש ב-Extra Javadoc options את השורה הבאה וללחוץ על כפתור Finish:
-tag requires:a:"Requires:" -tag modifies:a:"Modifies:" -tag effects:a:"Effects:"
- התנהגות ברירת המחדל של פעולות assert היא disabled (הבדיקות לא מתבצעות). כדי לאפשר את הידור וביצוע פעולות assert, יש לבצע ב-Eclipse את הפעולות הבאות: 1. מתפריט Run לבחור Debug Configurations , 2. בחלון שנפתח, לעבור ללשונית Arguments, 3. בתיבת הטקסט VM arguments לכתוב ea–, 4. ללחוץ על כפתור Debug.

הנחיות להגשת תרגילי בית

- תרגילי הבית הם חובה.
- ההגשה בזוגות בלבד.
- עם סיום פתירת התרגיל, יש ליצור קובץ דחוס להגשה המכיל את:
  - כל קבצי הקוד והתיעוד.
  - פתרון לשאלות ה"יבשות" בקובץ Word או PDF **בודד**. על הקובץ להכיל את שמות ומספרי תעודות הזהות **של שני הסטודנטים** המגישים.
- הגשת התרגיל היא **אלקטרונית בלבד**, דרך אתר הקורס **ע"י אחד מבני הזוג בלבד**.
- יש להגיש את הקבצים בנפרד (כלומר לא בzip).

- תרגיל שיוגש באיחור וללא אישור מתאים (כגון, אישור מילואים), יורד ממנו ציון באופן אוטומטי לפי חישוב של 2 נקודות לכל יום איחור.
- על התוכנית לעבור קומפילציה. על תכנית שלא עוברת קומפילציה יורדו 30 נקודות.

מועד ההגשה:
30.12.2018

המטרות של תרגיל בית זה הן:

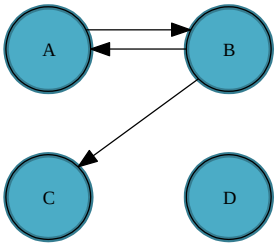
- להתנסות בתכן ADT (Abstract Data Type), בכתיבת מפרט עבורו, במימושו ובכתיבת קוד המשתמש בו.
- לתרגל ביצוע בדיקות יחידה בעזרת JUnit.
- התנסות בGenerics

הצגת הבעיה

בתרגיל בית זה, תיצרו הפשטה שתאפשר להריץ את אלגוריתם DFS (חיפוש לעומק בגרף) על גבי גרף מכוון. לשם כך תזדקקו להפשטה עבור גרף, המייצג את הקישוריות בין נקודות, להפשטה עבור מסלול, המייצג את מחיר המעבר דרך אוסף נקודות בגרף בסדר מסוים, לאלגוריתם DFS אשר רץ עם נקודת התחלה (ואופציונלית נקודת סיום) ולאלגוריתם אשר מודד ומשווה מסלולים בהנתן כמות הקשתות האחוריות בהם (עוד על כך בהמשך).

את ההפשטה עבור גרף ועבור מציאת מסלול קצר ביותר יהיה עליכם לתכנן בעצמכם.
את ההפשטה עבור גרף תאלצו לתכנן בעצמם ובנוסף גם את מימוש האלגוריתמים.
ההפשטה עבור מסלול נתונה לכם בממשק Path. ה-test driver המסופק, שאת מימושו תצטרכו להשלים, יבצע בדיקות, שכל אחת מהן תורכב מרשימת פקודות. פקודות אלה יתקבלו מאמצעי קלט (המקלדת/קובץ) ותוצאותיהן תשלחנה לאמצעי פלט (המסך/קובץ).

שאלה 1 (60 נקודות) *גרף מכוון (directed graph)* מורכב מאוסף של *צמתים (nodes)*, שחלקם עשויים להיות מקושרים בעזרת *קשתות (edges)*. לכל קשת יש כיוון, כלומר, עשוי להתקיים מצב, כמו בדוגמה הבאה, בו קיימת קשת המקשרת בין הצומת B לצומת C, אך לא קיימת קשת המקשרת בין הצומת C לצומת B.



בתרגיל זה נניח כי לא יכולה להיות יותר מקשת אחת המקשרת צומת מסוים לצומת אחר (אך יכולה, כמובן, להיות קשת בכיוון ההפוך).
הבנים (children) של B הם הצמתים שאליהם יש קשת מ-B. בדוגמה הנ"ל, הבנים של B הם A ו-C. *האבות (parents)* של B הם הצמתים שיש קשת מהם ל-B. בדוגמה הנ"ל, האב היחיד של B הוא A.

בשאלה זו תעסקו ביצירת טיפוס נתונים מופשט עבור גרף מכוון בעל משקלות (מחירים) לצמתים ובבדיקתו.

א.
עליכם להתחיל בכך שתחליטו על הפעולות שעל הפשטה זו לכלול. לשם כך, ניתן להיעזר באוסף הפקודות האפשרי בקובצי הבדיקה, המוגדר בנספח בסוף התרגיל. כתבו מפרט עבור ההפשטה שבחרתם, כולל פסקאות @requires, @modifies ו-@effects. את המפרט יש לרשום בקובץ בשם Graph.java.

הנחיות:
1. אובייקט שהוא מופע של Graph צריך לאפשר אכסון צמתים מטיפוס כלשהו. בפרט, צמתים אלה יכולים להיות טיפוסים מורכבים שמכילים פונקציונליות נוספת (למשל, מחשבים ברשת תקשורת).
2. מופע של Graph לא צריך לטפל ולא להיות מודע למחירי הצמתים. מחירים אלו יאוכסנו בתוך הצמתים עצמם.

להגשה ממוחשבת: המפרט של המחלקה Graph.
להגשה "יבשה": תיעוד חיצוני המסביר את השיקולים בבחירת הפעולות השונות של ההפשטה עבור גרף. הסבירו מדוע אוסף פעולות זה נראה לכם מספק לפתרון הבעיה הנתונה.

ב.
ממשו את המפרט שיצרתם בסעיף א'. בעת המימוש יש לזכור כי אלגוריתם DFS יעשה שימוש במימוש זה. אנו מעוניינים במימוש של Graph שהאלגוריתם DFS יתבצע עליו בסיבוכיות חישוב סבירה. אלגוריתם למציאת DFS משתמש באופן תכוף בפעולה של מציאת רשימת הבנים של צומת מסוים. עליכם לרשום מימוש שיבצע פעולה זו בזמן קבוע. גם הפעולות לבניית גרף צריכות להתבצע בזמן סביר. עם זאת, ראשית עליכם לדאוג לתכן נכון ורק לאחר מכן לביצועים טובים.

יש לרשום abstraction function ו-representation invariant בתוך שורות הערה בקוד של Graph. בנוסף, יש לממש מתודת checkRep() לבדיקת ה-representation invariant ולקרוא לה במקומות המתאימים בקוד.

הנחיות:
1. ב-Java Documentation ניתן למצוא את פירוט סיבוכיות החישוב של פעולות שונות של כל אחד מהמכלים הקיימים בשפת Java. ניתן להשתמש בנתונים אלה לשם הערכת סיבוכיות החישוב של מימושים שונים.
2. למרות שהפלט עבור הבדיקות של גרף מוגדר לעתים כרשימה של צמתים לפי סדר אלפביתי, אין זה אומר שהמימוש צריך להחזיר או להכיל צמתים לפי סדר זה. ניתן,

לחילופין, למיין את רשימת הצמתים לפני הצגתה. לשם כך ניתן להשתמש במתודה sort() של המחלקה java.util.Collections.

להגשה ממוחשבת: מימוש המחלקה Graph.
להגשה "יבשה": תיעוד חיצוני המסביר את השיקולים שהובילו למימוש הנבחר. הציעו **במילים** מימוש נוסף והשוו אותו לזה שבחרתם.

ג.
בדיקת המימוש של Graph תתבצע בעזרת test driver, ששלד מימוש שלו נתון במחלקה TestDriver. מחלקה זו קוראת אוסף פקודות המתקבלות ממקור קלט (למשל, קובץ או המקלדת), מפעילה אותן, ושולחת את התוצאות לאמצעי פלט (למשל, קובץ או המסך). עליכם להשלים את שלד מימוש זה במקומות המסומנים. יש להשתמש בשדה output לצורך ביצוע הפלט. לצורך בדיקת המימוש יש להשתמש במופעים של המחלקה הנתונה WeightedNode עבור צמתים בגרף. המבנה התחבירי של אוסף הפקודות החוקיות ותוצאותיהן מוגדר בנספח בסוף התרגיל.

נתונה המחלקה ScriptFileTests המשתמשת ב-JUnit ובמחלקה TestDriver לביצוע בדיקות של Graph. המחלקה עוברת על כל הקבצים בתיקייה בה היא נמצאת ושולחת את כל הקבצים בעלי סיומת test, אחד אחרי השני, ל-TestDriver. המחלקה מכוונת את הקלט של TestDriver לקובץ בעל שם זהה וסיומת actual. ומשווה לקובץ התוצאות הנדרשות בעל שם זהה וסיומת expected.

עליכם לרשום קובצי קלט לבדיקות קופסה שחורה של המימוש של Graph. על כל קובצי הבדיקה להיות בעלי הסיומת test ולכל אחד מהם צריך להיות קובץ נוסף בעל שם זהה לשם קובץ הבדיקה ובעל הסיומת expected שיכיל את הפלט הצפוי. תנו לקובצי הבדיקה שתיצרו בסעיף זה שמות משמעותיים. נתונים מספר קובצי בדיקה לדוגמה.

נתונה המחלקה GraphTests היורשת מהמחלקה ScriptFileTests כדי לבצע בדיקות קופסה שחורה בהתאם לקובצי הקלט המתאימים. עליכם להוסיף למחלקה GraphTests בדיקות קופסה לבנה ל-Graph. יש לרשום בדיקות בעזרת תחביר JUnit רגיל ולא להשתמש בקובצי קלט, כמו שנעשה עבור בדיקות קופסה שחורה.

בסעיף זה אין צורך לבצע בדיקות של האלגוריתם DFS (שעדיין לא קיים בשלב זה).

להגשה ממוחשבת: קובצי הקלט לבדיקות קופסה שחורה שכתבתם ותוצאות הרצתם (קובצי test, expected ו-actual). כמו כן, המחלקות שכתבתם לבדיקות קופסה לבנה ותוצאות הרצתן.
להגשה "יבשה": תיעוד חיצוני המסביר את השיקולים בבחירת הבדיקות הנ"ל ומדוע הן מספקות.

ד.
במחלקות ScriptFileTests ו-TestDriver נעשה שימוש במחלקה java.nio.file.Path בשמה המלא. מדוע לא מתבצע לה import בכותרת המחלקות ולאחר מכן שימוש בשם המקוצר Path?

להגשה "יבשה": תשובה לשאלה הנ"ל.

שאלה 2 (45 נקודות)

בשאלה זו תממשו ותבדקו אלגוריתם DFS (עם שינוי קל שלנו) ותשוו בין מסלולי DFS שונים.

DFS (ראשי תיבות של Depth-first search), הינו אלגוריתם המשמש למעבר על גרף או לחיפוש בו.

אינטואיטיבית, האלגוריתם מתחיל את החיפוש מצומת שרירותי בגרף ומתקדם לאורך הגרף עד אשר הוא נתקע, לאחר מכן הוא חוזר על עקבותיו עד שהוא יכול לבחור להתקדם לצומת אליו טרם הגיע. דרך פעולת האלגוריתם דומה במידת מה לסריקה שיטתית של מבוך.

רוב האלגוריתמים של DFS, מבצעים את הבחירה בילד מסויים על פי סדר אלפביתי (ילד B לפני ילד D), אך אצלנו נבחן את המחיר של צומת ונבחר את הילד בעל המחיר הגבוה ביותר. אם קיימים שני ילדים עם אותו המחיר, נבחר את האחד עם ערך אלפביתי גבוה יותר.

האלגוריתם בפסאודו קוד הוא:

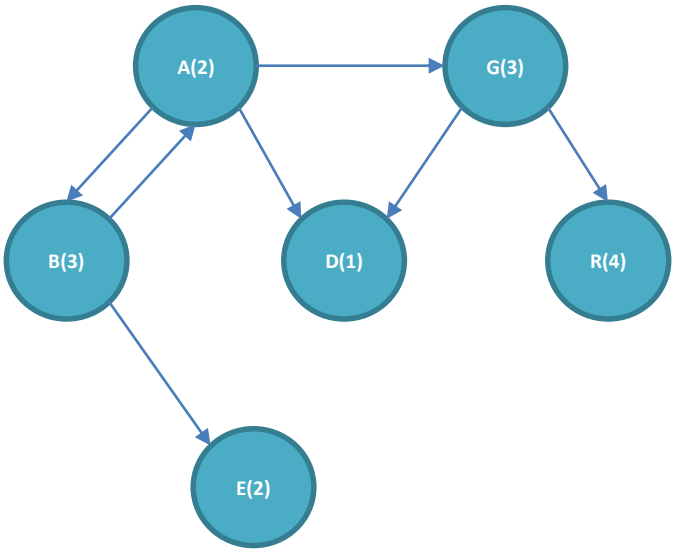
```
// INITIALIZATION
// Linked list of the nodes we already visited during our execution.
LinkedList visited = {}
// sets all color to white
Color(all Nodes, White)


// Return a DFS output: linked list
Algorithm DFS(Node start, Node end) {

   // The priority queue contains nodes with priority equal to the cost of the nodes. If
   // two nodes shares the same cost, choose them by alphabetical order (higher one
   // first). Otherwise, choose randomaly
  PriorityQueue childs = child_nodes

  // add the starting Node to the visited linked list
  visited.add(start)

   // each node is marked by one of three colors. color[v] - the color of node v
   //o white - an unvisited node
```

```
 //o gray - a visited node
 //o black - a node whose adjacency list has been examined
 //completely (whose descendants were all searched).
Color(start, grey)

if (start = end)
      return True

for each child c of childs {
      if (child not in visited){
              if (DFS(child, end) = true) {
                      return True
      }
 Color(start, Black)
 return False
```

<u>הערה</u>: צריך לתמוך גם במצב בו לא קיים end node, ואז נעבור על כל הצמתים שניתן להגיע אליהם מהצומת ממנו התחלנו.

במהלך האלגוריתם נעשה שימוש בתור עדיפויות. זהו מבנה נתונים המאפשר להכניס איברים ולהוציא איבר בעל העדיפות הגבוהה ביותר (במקרה שלנו, ככל שמחיר המסלול נמוך יותר, כך עדיפותו גבוהה יותר). יש לשים לב כי המחלקה java.util.PriorityQueue לא מאפשרת להכניס באופן ישיר את העדיפויות של איבריה. במקום, היא משתמשת לצורך השוואת עדיפויות האיברים במתודה compareTo() של הממשק java.util.Comparable.

א.
נתון הגרף הבא:



להגשה "יבשה": תיאור מילולי כללי של אופן פעולת האלגוריתם הנתון. בנוסף, הדגימו את שלבי פעולת האלגוריתם למציאת מסלול DFS בגרף הנתון עבור הקלטים שיוצגו בהמשך. בכל שלב יש לרשום מה מתבצע ואת ערכי visited, והצבעים השונים של הצמתים. אתם נדרשים לבצע זאת עבור הקלטים הבאים:

- $DFS\big(A(2), D(1)\big)$
- $DFS(B(3))$

ב. רשמו מפרט עבור מחלקה בשם DfsAlgorithm וממשו מחלקה זו. המחלקה תכיל מתודה למציאת מסלול DFS עבור צומת התחלה וסיום/צומת התחלה בלבד.
ניתן להניח כי צבעי הצמתים הם תקינים.
שימו לב, כי צבעי הצמתים בגרף צריכים לחזור להיות לבנים לאחר הרצת האלגוריתם.

טיפ: אם אתם רוצים להשתמש כבר פה ב בNodeCountingPath על מנת לייצג מסלול, זה אפשרי וזה יכול לחסוך לכם את ההתעסקות לאחר מכן.

להגשה ממוחשבת: המחלקה DfsAlgorithm

ג. כעת, נרצה להשתמש באלגוריתם DFS שכתבנו על מנת לבדוק האם קיימים מעגלים בגרף. אחת הדרכים לבדוק מעגלים היא באמצעות "קשתות אחריות" – כלומר, קשתות המחברות צאצא עם אב קדמון.
בהנתן האלגוריתם שמימשתם בסעיף ב', כיצד ניתן לבדוק האם קיים מעגל בגרף? מה התנאי לכך?
עדכנו את מחלקת DFS כך שעבור הרצה של האלגוריתם, יעודכנו הצמתים במידע החדש.
שימו לב שעליכם לעדכן גם את WeightedNode בהתאם (וגם את NodeCountingPath במידה והשתמשתם בו והתאמתם אותו). חשוב לציין, שהרצת DFS על הגרף לאחר השינוי, אמורה להניב את אותו המסלול מנקודה כלשהי A לנקודה B.
אין צורך להגיש שני קבצים נפרדים, פשוט עדכנו את המחלקות הקיימות.

להגשה ממוחשבת: WeightedNode, NodeCountingPath (אופציונלי למי שהשתמש)

ד. עליכם לרשום עבור מחלקה בשם PathFinder ולממש מחלקה זו. המחלקה תכיל מתודה למציאת מסלול **DFS** בעל מספר צמתים קטן ככל האפשר, כאשר קשתות אחוריות נספרות "כצומת" גם הן. שימו לב שרק קשתות אחוריות רלוונטיות למסלול נספרות. נקרא למספר זה **מחיר** של מסלול.

לדוגמה: אם קיים מסלול בין A לבין B, אשר מכיל 5 צמתים, כאשר מצומת אחד ישנה קשת אחורית אחת, אזי למסלול יינתן מחיר 6.

על המתודה לקבל אוסף צמתי התחלה ואוסף של צמתי סיום ולמצוא מסלול בעל מספר צמתים מינימלי ככל האפשר, כאשר קשתות נספרות "כצומת" גם הן, כך שהצומת הראשון בו הוא אחד מצמתי ההתחלה ושהצומת האחרון בו הוא אחד מצמתי הסיום.

על המחלקה PathFinder לתמוך בצמתים מסוג WeightedNode כלשהו. עליה לקבל את מספר הצמתים + מספר קשתות אחוריות בעזרת הממשק Path. כדי ליצור חוסר תלות בטיפוס הצמתים הספציפי, ניתן לתכנן את המתודה כך שתקבל מסלולי התחלה (מנוונים) ומסלולי סיום (מנוונים) במקום צמתי התחלה וצמתי סיום.

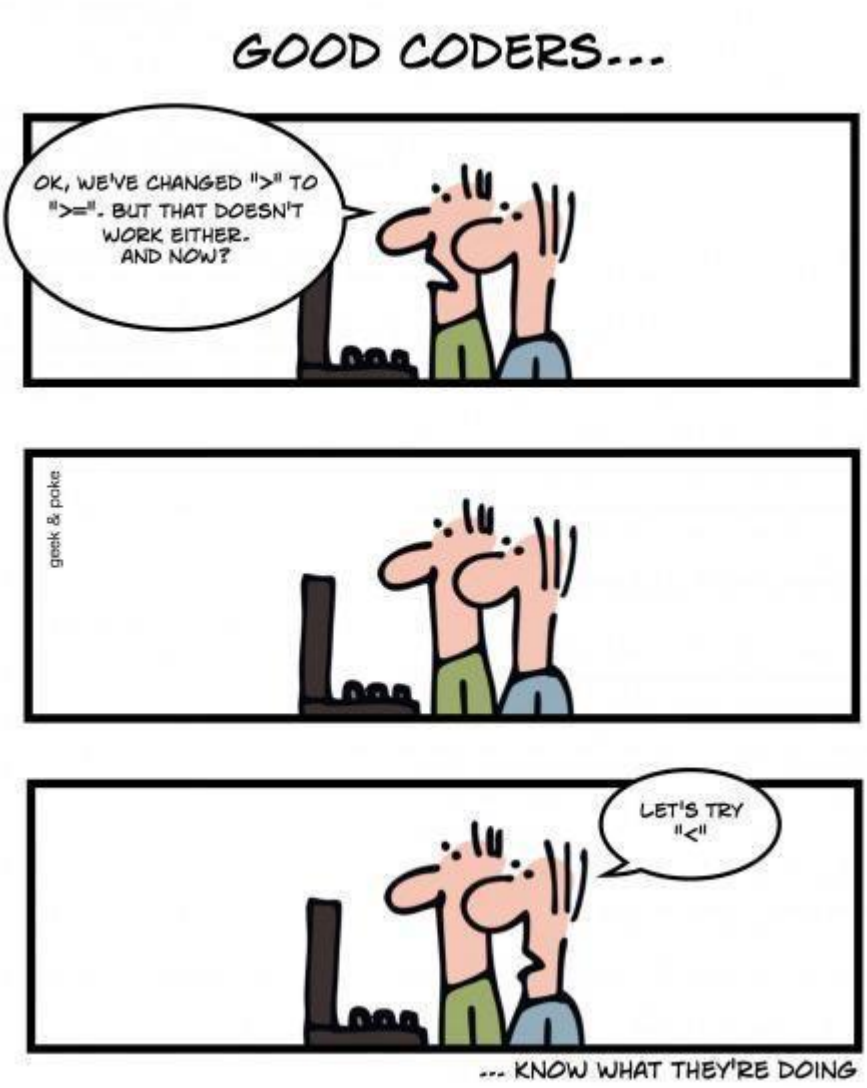
טיפ: מומלץ להסתכל בNodeCountingPath (ולהתאים אותה כך שתתמוך גם בספירת קשתות אחוריות. שימו לב לעדכן גם את התיעוד!) ולייצג באמצעותו מסלולים.

להגשה ממוחשבת: המחלקה PathFinder, NodeCountingPath (יתקבל גם מימוש ללא NodeCountingPath, אבל במידה והשתמשתם/עדכנתם אותה, אנא הגישו גרסה חדשה שלה).

ה.
הרחיבו את סביבת הבדיקה שיצרתם בשאלה 1ג' עבור Graph כך שתבדוק גם DfsAlgorithm ואת PathFinder ע"י הוספת קובצי קלט לבדיקות קופסה שחורה ובדיקות JUnit לבדיקות קופסה לבנה.

להגשה ממוחשבת: קובצי הקלט לבדיקות קופסה שחורה שכתבתם ותוצאות הרצתם (קובצי test, expected ו-actual). כמו כן, בדיקות קופסה לבנה שכתבתם ותוצאות הרצתן. להגשה "יבשה": תיעוד חיצוני המסביר את השיקולים בבחירת הבדיקות הנ"ל ומדוע הן מספקות.

<u>הערה</u>: בתרגיל 105 נקודות, כאשר ניתן לקבל בתרגיל 100 לכל היותר. הבונוס הוא לתרגיל בית הזה ספיציפי ואינו משפיע על תרגילי הבית האחרים/מבחן.


GOOD CODERS...
OK, WE'VE CHANGED ">" TO ">=". BUT THAT DOESN'T WORK EITHER. AND NOW?
geek & poke
LET'S TRY "<"
... KNOW WHAT THEY'RE DOING

נספח: מבנה קובץ פקודות לבדיקה

קובץ בדיקה הוא קובץ טקסט המורכב מתווים אלפאנומריים והמכיל פקודה אחת בכל שורה. כל שורה מורכת ממילים המופרדות זו מזו ברווחים או בטאבים. המילה הראשונה בכל שורה היא שם הפקודה ושאר המילים הן ארגומנטים לפקודה זו. שורות המתחילות בסולמית (#) הן שורות הערה והן יועברו לאמצעי הפלט ללא שינוי בעת הרצת הבדיקה. שורות ריקות יגרמו להדפסת שורות ריקות באמצעי הפלט.

קובץ בדיקה מטפל בצמתים מטיפוס WeigthedNode ובמסלולים מטיפוס WeightedNodePath. לכל WeigthedNode יש שם ומחיר. לאחר שצומת כזה נוצר, ניתן להתייחס אליו בשמו, ובכל התייחסות אליו בפלט, הוא יוצג בשמו. גם לכל גרף יש שם, שההתייחסות אליו היא באותו אופן.

רשימת הפקודות והפלט שהן מחזירות היא:

CreateGraph *graphName*
יוצרת גרף חדש בשם *graphName*. הגרף מאותחל להיות ריק (ללא צמתים וללא קשתות). פלט הפקודה הוא:

created graph *graphName*

CreateNode *nodeName cost*
יוצרת צומת חדש בשם *nodeName* עם מחיר שהוא המספר השלם והלא שלילי *cost*. לאחר יצירת צומת בעזרת פקודה זו, ניתן להתייחס אליו בשמו. פלט הפקודה הוא:
created node *nodeName* with cost *cost*

AddNode *graphName nodeName*
מוסיפה את הצומת ששמו הוא המחרוזת *nodeName* לגרף ששמו *graphName*. פלט הפקודה הוא:

added node *nodeName* to *graphName*

AddEdge *graphName parentNode childNode*
יוצרת קשת בגרף *graphName* מהצומת *parentNode* לצומת *childNode*. פלט הפקודה הוא:

added edge from *parentNode* to *childNode* in *graphName*

ListNodes *graphName*
לפקודה זו אין השפעה על הגרף. פלט הפקודה מתחיל ב:

graphName contains:
ואחריו, באותה שורה, רשימה מופרדת ברווחים של שמות כל הצמתים בגרף *graphName*. על הצמתים להופיע בסדר אלפביתי. קיים רווח יחיד בין הנקודתיים לשם הצומת הראשון.
הערה: נקבל גם רשימת הצמתים פי compareTo שמומש בWeightedNode. שימו לב שבמידה ותממשו זאת ככה – אזי הexpected שלכם ייראה שונה מזה שקיים באתר.

ListChildren *graphName parentNode*

לפקודה זו אין השפעה על הגרף. פלט הפקודה מתחיל ב:

the children of *parentNode* in *graphName* are:

ואחריו, באותה שורה, רשימה מופרדת ברווחים של שמות הצמתים שיש אליהם צומת מ-*parentNode* בגרף *graphName*. על הצמתים להופיע בסדר אלפביתי. קיים רווח יחיד בין הנקודתיים לשם הצומת הראשון.

הערה: נקבל גם רשימת הצמתים פי compareTo שמומש בWeightedNode. שימו לב שבמידה ותממשו זאת ככה – אזי הexpected שלכם ייראה שונה מזה שקיים באתר.

FindPath *graphName from1* [*from2* [*from3* ...]] -> *to1* [*to2* [*to3* ...]]

הסימן <- מופרד משמות הצמתים ברווח משני צדדיו. הסוגריים המרובעים מציינים איברים שאינם חובה (הסוגריים המרובעים לא מופיעים בקובץ הקלט). ניתן לרשום מספר כלשהו של צמתי *from* ומספר כלשהו של צמתי *to*. לפקודה זו אין השפעה על הגרף. היא מוצאת ויוצרת פלט של מסלול בעל מחיר קטן ביותר בין כל המסלולים האפשריים בין אחד מצמתי ה-*from* לאחד מצמתי ה-*to* בגרף *graphName*. פלט הפקודה מתחיל ב:

found path in *graphName*:

ואחריו, באותה שורה, רשימה מופרדת ברווחים של שמות הצמתים לפי סדר המעבר בהם מאחד מצמתי ה-*from* לאחר מצמתי ה-*to*. אם לא קיים מסלול, הפלט יהיה:

no path found in *graphName*

במידה ולא קיים מסלול עבור אלגוריתם הDFS, הפלט יהי

dfs algorithm output <graph_name> <start_node> -> <end_node>: no path was found

לדוגמה, עבור הקלט הבא:

```
CreateNode n1 5
CreateNode n3 1
CreateGraph A
AddNode A n1
CreateNode n2 10
AddNode A n2
CreateGraph B
ListNodes B
AddNode A n3
AddEdge A n3 n1
AddNode B n1
AddNode B n2
AddEdge B n2 n1
AddEdge A n1 n3
AddEdge A n1 n2
ListNodes A
ListChildren A n1
AddEdge A n3 n3
ListChildren A n3
```

DfsAlgorithm A n1
FindPath A n3 -> n2

פלט תקין יהיה :

created node n1 with cost 5
created node n3 with cost 1
created graph A
added node n1 to A
created node n2 with cost 10
added node n2 to A
created graph B
B contains :
added node n3 to A
added edge from n3 to n1 in A
added node n1 to B
added node n2 to B
added edge from n2 to n1 in B
added edge from n1 to n3 in A
added edge from n1 to n2 in A
A contains: n1 n2 n3
the children of n1 in A are: n2 n3
added edge from n3 to n3 in A
the children of n3 in A are: n1 n3
dfs algorithm output A n1: n1 n2 n3
found path in A: n3 n1 n2 with cost 5

alternative for found path (counting only back edges that on our way to the destination):
found path in A: n3 n1 n2 with cost 3

הפלט עבור קלט לא תקין אינו מוגדר. ניתן להניח שהקלט הוא לפי התחביר החוקי בלבד.

אם הרצת הקלט גורמת לזריקת חריגה, תשלח הודעה מתאימה לאמצעי הפלט. יכולת זו כבר ממומשת בשלד של המחלקה TestDriver.