Object Oriented Programming and Design

Homework 2

דניאל לוי 315668129

לידור אלפסי 307854430

שאלה 1

א. לשם המחלקה Graph בחרנו בפעולות שייצגו לנו את הגרף בצורה הבסיסית והטובה ביותר. מטרת המחלקה היא אך ורק ייצוג גרף, ולכן הגבלנו לשימושים אלה.
getName- מתודה שתאפשר לנו לזהות את שם הגרף. שימושית בעיקר לצרכי דיבוג.
addNode- הוספת צומת לגרף. פעולה הכרחית לשם בניית גרף.
addEdge- הוספת קשת לגרף. פעולה הכרחית לשם בניית גרף.
בלעדיהן המחלקה תייצג גרף ריק, שלא נותן לנו שום משמעות לקיום שלו.
getChildren- פעולה מסוג זה נדרשת לשם פעולות על הגרף, כמו בזמן חיפוש הדרך הקצרה ביותר בגרף.
getNodes- פעולה נדרשת על מנת לאפשר תצוגה של הגרף בהמשך, וגם למטרות דיבוג.
בשתי הפעולות החלטנו להעביר איטרטור לרשימות של הצמתים, או הבנים של צומת מסוימת. כך אנחנו נותנים למשתמש את הגישה למידע שהוא צריך, מבלי לחשוף את כל מבנה הנתונים שלנו בפניו.
לא ראינו צורך בהוספת פעולות נוספות. מימוש המחלקה כמחלקה גנרית, אשר לא אמורה להניח דבר על הצמתים(ועל המחיר שלהם לצורך העניין), פישט את הצרכים מהמחלקה.

ב. בחרנו לממש את המחלקה כמבנה נתונים מסוג Map, אשר ממפה צומת(מחלקה גנרית Node) אל Set שמייצג את צמתי הבנים של הצומת.
העדפנו מבנה נתונים שיעניק לנו גישה מהירה וקלה לרשימת הבנים של כל צומת, ולכן Map הייתה אפשרות טובה לכך. בנוסף לכך, רשימת הבנים של צומת מסוימת היא בעצם קבוצה בה כל צומת בן יכולה להופיע רק פעם אחת, ולכן בחרנו ב-Set לשם מימוש רשימת הבנים של כל צומת בגרף.
לא הנחנו שום סדר על הצמתים במחלקה עצמה, והעדפנו שלא להוסיף לה פעולות סבוכות אשר יוסיפו לסיבוכיות הפעולות שלה. אילוץ כמו החזרת רשימת הצמתים כרשימה אלפביתית הוא אילוץ שמתאים למקרים ספציפיים בלבד. למשל, בתרגיל זה נדרש רק לצורך ההדפסה בטסטים, אך זה לא נצרך באלגוריתם החיפוש של המסלול הקצר ביותר. לכן העדפנו לתת למשתמש רשימה שאינה מסודרת לפי סדר כלשהו, ושהוא ידאג לסידורם ברשימה לאחר מכן. כך הצלחנו גם לחסוך מסיבוכיות הרצת פעולות הגרף.
לחילופין, יכולנו לממש את מחלקת ה-Graph כList שמכיל Lists בפני עצמם. כל List פנימי היה מייצג לנו את הצומת של הגרף במקום הראשון, ואחריו כל

הבנים שלו. כמובן שזהו מימוש סבוך יותר, בעל סיבוכיות פעולות גדולה יותר, ואשר דורש בדיקות רבות על קיום הRep. Invariant. זאת לעומת המימוש בו השתמשנו, שבו ה-Rep. Invariant נשמר ברובו המוחלט על ידי מבני הנתונים עצמם. Map לא יאפשר הוספת מופע כפול של צומת, אלא רק דריסה של אותו הצומת בגרף, ו-Set לא מאפשר הכנסה חוזרת של צומת מעצם ההגדרה שלו. לכן לא נצטרך לבדוק בכלל כפילויות של מופעי הצמתים.

ג. מבחינת בדיקות קופסה שחורה- היה חשוב לנו לבדוק מקרים בסיסיים של הוספת צמתים וקשתות בעומס שהולך וגדל. רצינו לראות שלא קורה מצב בו בזמן הוספת צמתים או קשתות יש משהו שפוגם בתקינות הגרף. זאת בנוסף להדפסות שעשינו בעיקר בטסט הכולל הוספה מסיבית של צמתים וקשתות. בבדיקות הקופסה הלבנה, מטרתנו הייתה להגיע למקרי הקצה, שמהם בעיקר זרקנו שגיאות או טיפלנו באופן נקודתי. למשל, בזמן שליחת name=null ביצירת גרף חדש, החלטנו במימוש שהשם יוגדר כמחרוזת ריקה במקום להכשיל את כל הפעולה. ולעומת זאת, בזמן הוספת צומת או קשת רצינו לכסות את המצב בו המשתמש שולח node=null. פעולה שכזו תזרוק אצלנו שגיאה, ורצינו לוודא שזה אכן קורה.

ד. ביצוע Import למחלקה היה יוצר לנו בלבול אל מול java.nio.file.Paths שנמצא בשימוש גם כן בקובץ. באופן כללי עדיף לחשוב על שימוש בדומיין המלא של המחלקה, בכדי להקטין את הסיכוי להתנגשות בין שמות, ובכדי להוסיף להבנת הקוד.
בקובץ הזה נכנסת עובדת השימוש בPaths וPath במקביל. אם לא נבצע ביניהם הפרדה ברורה אז נגדיל את הסיכוי שלנו להתבלבל בין השניים.

שאלה 2

א. אופן פעולת האלגוריתם הנתון-

- יצירת Map שממפה צמתים אל מסלולים(Path) אשר מובילים אל הצמתים האלה. כעת כל מסלול הוא רק צומת אחת.
- יצירת תור עדיפויות active שיכיל את כל צמתי ההתחלה, עם עדיפות שהיא מחיר כל צומת. בפועל יכיל את המחיר הנמוך ביותר להגעה לכל צומת בו.
- יצירת סט finished שיכיל צמתים שכבר בחנו את הבנים שלהם(בהמשך).
- לולאה: כל עוד תור העדיפויות active לא ריק
  - הוצאת הצומת עם המחיר המינימלי להגעה אליו(כרגע) queueMin מהתור.
  - מחפשים את המסלול המתאים אליו(עם המחיר הנמוך) ב-Map שיצרנו- queueMinPath
  - אם queueMin שייך ל-goals, אז מצאנו כבר את הדרך הקצרה ביותר אליו.
    - נחזיר את הדרך הזו(queueMinPath) למשתמש.
  - לולאה: ריצה על כל בן c של הצומת queueMin
    - נגדיר cpath כדרך queueMinPath בצירוף הצומת c
    - אם c לא נמצא ב-finished וגם לא בתור העדיפויות active
      - נכניס ל-Map של הדרכים מיפוי מ-c ל-cpath
      - נכניס את c לתור העדיפויות active עם עדיפות ששווה למחיר cpath
    - הכנסת queueMin לסט ה-finished, כי סיימנו לעבור עליו ולסרוק את הבנים שלו.
- החזרת אינדיקציה שאין בכלל מסלול מ-starts אל goals

אנחנו שומרים בכל פעם את המסלול הקצר ביותר והמחיר הנמוך ביותר אל צומת מסוימת בעזרת paths ו-active. איך נדע שלא יבוא מחיר יותר זול בהמשך? פשוט אנחנו בכל פעם בודקים את הדרך אל הצמתים על ידי הדרך המינימלית עד עכשיו. אם הצומת העכשווית תופיע בהמשך, אז סימן שבהכרח המחיר אחר כך יהיה גבוה יותר(כי המחיר שמתווסף אותו הדבר, והגענו מדרך יותר קלה בעבר).
כך האלגוריתם החמדן בעצם בודק את כל האפשרויות על ידי הדרכים הקצרות ביותר בכל רגע נתון.

הדגמת שלבי פעולת האלגוריתם-

- הכנסת A אל ה-Map של paths
  הכנסת A אל תור העדיפויות active עם מחיר 2
  איתחול finished כסט ריק
  (paths={A,[A]} , active=[(A,2)] , finished= {})
- הוצאת הצומת A מתור העדיפויות כ-queueMin
  לקיחת המסלול [A] כ-queueMinPath
  ריצה על הבנים של A
  נתחיל מ-B
  - נגדיר cpath=[A,B]
  - כמובן ש-B לא נמצא ב-finished או בactive
  - נכנס ל-paths את המיפוי- {B, [A,B]}
  - נכניס ל-active את צומת B עם עדיפות 3

  איטרציה על C(הבן השני של A)
  - נגדיר cpath=[A,C]
  - C לא נמצא ב-finished או ב-active
  - נכניס ל-paths את המיפוי- {C, [A,C]}
  - נכניס ל-active את צומת C עם עדיפות 5

  הכנסת queueMin(הצומת A) אל הסט finished
  paths={A,[A]}{B,[A,B]}{C,[A,C]} , active=[(B,3)(C,5)] , finished= {A}
- הוצאת הצומת B מתור העדיפויות כ-queueMin(עם המחיר הנמוך-3)
  לקיחת המסלול [A,B] כ-queueMinPath
  ריצה על הבנים של B
  נתחיל מ-D
  - נגדיר cpath=[A,B,D]
  - D לא נמצא ב-finished או ב-active
  - נכניס ל-paths את המיפוי- {D,[A,B,D]}
  - נכניס ל-active את צומת D עם עדיפות 4

  איטרציה על C(הבן השני של B)
  - נגדיר cpath=[A,B,D,C]
  - C נמצא ב-active, ולכן נמשיך לאיטרציה הבאה

  הכנסת queueMin(הצומת B) אל הסט finished
  paths={A,[A]}{B,[A,B]}{C,[A,C]}{D,[A,B,D]} ,
  active=[(D,4)(C,5)] , finished= {A,B}

- הוצאת הצומת D מתור העדיפויות כ-queueMin(עם המחיר הנמוך 4)
לקיחת המסלול [A,B,D] כ-queueMinPath
ריצה על הבנים של D
יש בן יחיד-E
  - נגדיר cpath=[A,B,D,E]
  - E לא נמצא ב-finished או ב-active
  - נכניס ל-paths את המיפוי- {E,[A,B,D,E]}
  - נכניס ל-active את הצומת E עם עדיפות 8

הכנסת queueMin(הצומת D) אל הסט finished

paths={A,[A]}{B,[A,B]}{C,[A,C]}{D,[A,B,D]}{E,[A,B,D,E]} ,
active=[(C,5)(E,8)] , finished= {A,B,D}

- הוצאת הצומת C מתור העדיפויות כ-queueMin(עם המחיר הנמוך-5)
לקיחת המסלול [A,C] כ-queueMinPath
ריצה על הבנים של C
נתחיל מ-D
  - נגדיר cpath=[A,C,D]
  - D נמצא ב-finished, ולכן נמשיך לבן הבא

נמשיך עם הבן E
  - נגדיר cpath=[A,C,E]
  - E נמצא ב-active, ולכן נמשיך לאיטרציה הבאה

הכנסת queueMin(הצומת C) אל הסט finished

paths={A,[A]}{B,[A,B]}{C,[A,C]}{D,[A,B,D]}{E,[A,B,D,E]} ,
active=[(E,8)] , finished= {A,B,D,C}

- הוצאת הצומת E מתור העדיפויות כ-queueMin(עם המחיר הנמוך-8)
לקיחת המסלול [A,B,D,E] כ-queueMinPath

paths={A,[A]}{B,[A,B]}{C,[A,C]}{D,[A,B,D]}{E,[A,B,D,E]} ,
active=[] , finished= {A,B,D,C}

queueMin שהוא הצומת E נמצא ב-goals
לכן נחזיר את הדרך [A,B,D,E] למשתמש

ג. בביצוע בדיקות קופסה שחורה כיסינו כמה שיותר מקרים אפשריים בגבולות הפעולות שניתן לעשות. רצינו לבדוק מה קורה כאשר יש צומת עם לולאה עצמית, האם כשאין דרך מהצומת לעצמו אז נחזיר בכל מקרה רק את הצומת כדרך ריקה. רצינו לכסות אפשרויות שבהן יש מספר צמתי התחלה ומספר צמתי יעד. ולבסוף רצינו לוודא מה קורה בגרף עם חיבור מאסיבי בין הצמתים, ולבדוק בתרחיש בו יש המון פעולות עם מקור לכישלון הטסט.
בביצוע בדיקות קופסה לבנה התמקדנו במקרה הפרטי בו המשתמש שולח גרף שהוא בעצם null, ורצינו לוודא שכיסוי המקרה מתרחש כרצוי על ידי זריקת חריגה. בנוסף לכך רצינו לוודא פעולה בסיסית של חיפוש דרך בין ששני צמתים שלא מקושרים ביניהם, בכדי לוודא שאנחנו אכן מגיעים ל-return null במימוש פונקציית ה-findPath שלנו.

ד. Instruction coverage

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
homework2	69.2 %	1,253	557	1,810
homework2	69.2 %	1,253	557	1,810
homework2	69.2 %	1,253	557	1,810
TestDriver.java	76.1 %	643	202	845
NodeCountingPath.java	0.0 %	0	162	162
WeightedNodePath.java	48.2 %	79	85	164
WeightedNode.java	25.7 %	19	55	74
GraphTester.java	80.7 %	121	29	150
GraphScriptFileTester.java	90.7 %	156	16	172
Graph.java	96.3 %	105	4	109
PathFinder.java	97.0 %	130	4	134

Branch coverage

Element	Coverage	Covered Branches	Missed Branches	Total Branches
homework2	54.9 %	90	74	164
homework2	54.9 %	90	74	164
homework2	54.9 %	90	74	164
NodeCountingPath.java	0.0 %	0	24	24
TestDriver.java	69.7 %	46	20	66
WeightedNodePath.java	25.0 %	6	18	24
WeightedNode.java	0.0 %	0	8	8
GraphScriptFileTester.java	75.0 %	6	2	8
Graph.java	92.9 %	13	1	14
PathFinder.java	95.0 %	19	1	20
GraphTester.java		0	0	0

ה. אנחנו רואים כיסוי גבוה יותר של הפעולות.

בקוד שלנו הצלחנו להגיע לכיסוי רחב מאוד. כיסינו את כל הפקודות וההסתעפויות מלבד אלה שקיימים ב-checkRep, שם הAssert נספר כהסתעפות/פעולה צהובה אשר מן הסתם לא ניתן לבדוק אותה במלואה. זאת משום שה-Assert מוודא ששמרנו על מצב ה- .Rep Invariant. לא הצלחנו לכפות כישלון של ה-Assert מבחוץ, עקב הצלחה בשמירה על הדרוש מתוך המימוש.

הסיבה לכך שהכיסוי של הפעולות נראה גבוה היא העובדה שיש יותר פעולות מהסתעפויות בקוד שלנו. אי הכיסוי של הcheckRep משפיע בצורה יותר משמעותית על כיסוי ההסתעפויות מאשר כיסוי הפעולות. זה נכון גם לגבי שאר הקבצים, ולא רק אלה שבדקנו בטסטים ייעודיים.

יותר קל להשיג כיסוי של פעולות, כפי שניתן לראות מהמקרה שלנו. קיים קושי בכיסוי כל ההסתעפויות ממימוש חיצוני. וכאשר מגיע מצב בו הצלחנו לכסות הסתעפות, אז גם מספר הפעולות שקיימות בכיסוי עולה בהתאם. כיסוי של פעולות דורש פחות מקרי בוחן(טסטים) מאשר כיסוי הסתעפויות.

הסתעפויות הן נקודות יותר רגישות בקוד לדעתנו, והן דורשות יותר מקרי בוחן בכדי לאפשר כיסוי רחב יותר של האפשרויות. בכדי להבטיח את נכונות הקוד נרצה שכל ההסתעפויות האפשריות ייבדקו. גם אם מדובר בהסתעפות שולית שלא מורגשת בכיסוי פעולות בגלל שמדובר בשורת קוד אחת, אז זאת הסתעפות שיכולה לגרום לנו לבאג בקוד.