הפקולטה להנדסה

המעבדה להגנת סייבר

הוכחות באפס ידיעה ואפליקציות קריפטוגרפיות

בר אברהם דעבול

נדב יוסף זדה

פרויקט שנה ד' לקראת תואר ראשון בהנדסה

מנחה אקדמית: ד"ר מור וייס

ספטמבר 2023

תוכן עניינים

מבוא

במתמטיקה ובלוגיקה הוכחה היא סדרה סופית של טענות הנובעות זו מזו בעזרת כללי היסק, תוך שימוש בהגדרות, באקסיומות, ובידע קודם שהוכח קודם לכן, המראה שטענה מסוימת היא נכונה.
במציאות הוכחה היא לא אובייקט קבוע והיא יכולה להיות משהו שאפשר לחזות בו כמו למשל עדות בבית משפט.

הפרויקט שלנו עוסק בהוכחות באפס ידיעה ואפליקציות קריפטוגרפיות.
הוכחות באפס ידיעה מאפשרות להוכיח טענות, בצורה שבה לא חושפים מידע נוסף מלבד העובדה שהטענות נכונות. הוכחות מסוג זה הן כלי מרכזי בתכנון של פרוטוקולים קריפטוגרפיים.
ישנם יישומים רבים שמשתמשים בהוכחה באפס ידיעה, כגון: סכמת הזדהות (למשל עבור כניסה לחשבון אימייל), הוכחת התנהגות הגונה בפרוטוקולי חישוב מרובי משתתפים, אנונימיות בבלוקצ'יין ובמטבעות דיגיטליים (הסתרת פרטי הטרנסאקציה) והצבעה אלקטרונית.

מטרת הפרויקט היא מימוש פרוטוקולים להוכחות באפס ידיעה והשוואה ביניהם.
השלב הראשון של הפרויקט הוא למידת רקע תיאורטי על קריפטוגרפיה, על הוכחות באפס ידיעה ועל הפרוטוקולים הרלוונטיים. לאחר מכן, מתבצע תכנון המימוש של הפרוטוקולים, תוך בחינת דרכים שונות למימוש. בהמשך, נכתב קוד למימוש של הפרוטוקולים בתוכנה. לאחר המימוש, מתבצעים ניסויים שונים להשוואת הפרוטוקולים על דוגמאות קלטים שונות תחת פרמטרים שונים. בסופו של דבר, נציג את המסקנות שעולות מהממצאים.

רקע תיאורטי

הוכחות באפס ידיעה

מערכת הוכחה באפס ידיעה היא מערכת הוכחה אינטראקטיבית שמקיימת תכונה נוספת שנקראת אפס ידיעה [Zero-Knowledge (ZK)].

מערכת הוכחה אינטראקטיבית היא מערכת הוכחה שבה יש שני משתתפים (מוכיח ומוודאת) שמבצעים את שתי המשימות העיקריות של מערכת ההוכחה – לספק את ההוכחה ולוודא את התקפות שלה. במערכת הזאת יש אינטראקציה בין שני המשתתפים והיא מקיימת שני תנאים (שלמות ונאותות).

ראשית, נגדיר באופן פורמלי מהי מערכת הוכחה אינטראקטיבית [Interactive Proof (IP)]:

מערכת הוכחה אינטראקטיבית לשפה $L \in NP$ היא פרוטוקול בין זוג משתתפים P, ו-V PPT כך שמתקיימים שני התנאים הבאים:

- **Completeness:**

$$\forall x \in L \text{ and } \forall \text{ witness } w, \Pr\left[\left(P(x,w), V(x)\right) = 1\right] \geq \frac{2}{3}$$

- **Soundness:**

$$\text{if } x \notin L \text{ then for every } P^*, \Pr\left[\left(P^*(x), V(x)\right) = 1\right] \leq \frac{1}{3}$$

כלומר, מערכת ההוכחה צריכה לקיים שני תנאים – שלמות (Completeness) ונאותות (Soundness).

השלמות מבטאת את היכולת של המוכיח לשכנע את המוודאת בטענות נכונות. בפרוטוקול זה, דורשים שלכל מחרוזת בשפה L, הפלט של V על אותה מחרוזת באינטראקציה עם P יהיה 1 (כלומר V מקבל את הטענה - accept), בהסתברות של לפחות $\frac{2}{3}$.

הנאותות מבטאת את היכולת של המוודאת לא לקבל טענות שגויות. בפרוטוקול זה, דורשים שלכל מחרוזת שלא בשפה L, ולכל מוכיח כלשהו P^* הפלט של V על אותה מחרוזת באינטראקציה עם P^* יהיה 1 (כלומר V מקבל את הטענה - accept), בהסתברות של לא יותר מ- $\frac{1}{3}$.

ההגדרה הפורמלית של תכונת האפס ידיעה (ZK) משתמשת באלגוריתם סימולטור Sim. לכן, נגדיר כיצד מוגדר סימולטור.

סימולטור Sim הוא אלגוריתם PPT, שיש לו את אותו הקלט כמו זה של המוודאת V^*, אך הוא אינו משתתף בפרוטוקול. המטרה של הסימולטור היא לייצר את ה-view של המוודאת V^*, כלומר כל מה ש-V^* לומדת מהאינטראקציה. ה-view של V^* כולל את הקלט שלה, את האקראיות שלה (random coins) ואת כל ההודעות שהיא קיבלה מהמוכיח P. הסימולטור יכול לתקשר עם V^* על מנת לייצר את ה-view המסומלץ.

דרישת האפס ידיעה (ZK) היא שההתפלגות המשותפת של כל מרכיבי ה-view של V^* מהאינטראקציה עם P (ה-view מהריצה האמיתית), והתפלגות הפלט של הסימולטור (ה-view המסומלץ) קרובות חישובית/סטטיסטית/בעלות אותה התפלגות.

כעת, נגדיר מהי מערכת הוכחה באפס ידיעה [Zero-Knowledge Proof (ZKP)]:

מערכת הוכחה באפס ידיעה לשפה $L \in NP$ היא מערכת הוכחה אינטראקטיבית (IP) בין מוכיח P ומוודאת PPT V המקיימת את תכונת האפס ידיעה (ZK) הבאה:

$$\forall\ \boldsymbol{PPT\ V^*}\ \exists\ \boldsymbol{PPT\ simulator\ Sim\ s.t.}\ \forall \boldsymbol{x} \in \boldsymbol{L\ with\ witness\ w}:$$

$$\boldsymbol{Sim}(\boldsymbol{x}) \approx \boldsymbol{View_{V^*}}\big(\boldsymbol{P}(\boldsymbol{x},\boldsymbol{w}), \boldsymbol{V^*}(\boldsymbol{x})\big)$$

כלומר, מערכת הוכחה באפס ידיעה לכל שפה $L \in NP$ היא פרוטוקול בין זוג משתתפים P, ו-V *PPT* שמקיימת שלושה תנאים – שלמות (*Completeness*), נאותות (*Soundness*) ואפס ידיעה (*ZK*) (Goldreich, 2001).

סכמות התחייבות

סכמות התחייבות משמשות משתתפים להתחייב על ערך מסוים ועדיין לשמור אותו בסוד. בשלב מאוחר יותר ההתחייבויות נפתחות ומובטח שהפתיחה יכולה להניב רק ערך בודד שנקבע בשלב ההתחייבות (כמו שמירה של ערך במעטפה או כספת ופתיחה בשלב מאוחר יותר).

סכמת התחייבות Π היא פרוטוקול דו-שלבי בין מתחייב (*Commiter*) *C – PPT* לבין מקבלת (*Reciever*) *R – PPT*. באופן כללי, הפרוטוקול פועל באופן הבא:

<u>הקלט של הפרוטוקול:</u> למתחייב *C* יש ביט $b \in \{0,1\}$, המתחייב והמקבלת מקבלים את פרמטר הבטיחות 1^n.

<u>שלב ההתחייבות (*Commit phase*):</u> שלב אינטראקטיבי בין *C* ו-*R* (בין המתחייב והמקבלת). נוצרת התחייבות *c* שמכילה את 1^n (פרמטר הבטיחות), את האקראיות של *R* ואת כל ההודעות שנשלחות ל-*R*. כלומר, *c* הוא ה-*view* של *R*.

<u>שלב הפתיחה (*Decommit phase*):</u> *C* שולח ל-*R* את הביט *b* ומחרוזת (*decommitment string*) *dec*. *R* מוציאה כפלט "קבלה של *b*" (*"accept,b"*) או "דחייה" (*"reject"*).

סכמת ההתחייבות צריכה לקיים מספר תכונות:

- **<u>*Correctness* (נכונות)</u>**

  לכל $b \in \{0,1\}$, $n \in \mathbb{N}$, מתקיים: $\Pr[R\ outputs\ (accept, b)\ in\ \Pi] = 1$.
- **<u>*Hiding*</u>**

  כל *PPT* R^* לא לומדת כלום על *b* במהלך ההתחייבות. כלומר, R^* יכולה לנחש בהצלחה את *b* רק בהסתברות של $\frac{1}{2} + negl$.
- **<u>*Binding*</u>**

  לא קיים C^* שיכול לבצע *decommit* (לפתוח את סכמת ההתחייבות) ל-2 ערכים שונים. כלומר, C^* יכול לפתוח בהצלחה גם ל-0 וגם ל-1 רק בהסתברות זניחה (*negl*).

באופן אינטואיטיבי, שתי התכונות האחרונות של סכמת ההתחייבות מקיימות את הדרישות הבאות (בהתאמה):

1. <u>*Hiding*</u>: בסוף השלב הראשון המקבלת (R^*) לא משיגה שום ידע על הערך של המתחייב (C^*). זה חייב להתקיים גם אם המקבלת מנסה לרמות.
2. <u>*Binding*</u>: בהינתן ביצוע של השלב הראשון (חישוב ההתחייבויות), קיים לכל היותר ערך אחד שהמקבלת יכולה לקבל מאוחר יותר (בשלב השני) כפתיחה של ההתחייבות. זה חייב להתקיים גם אם המתחייב מנסה לרמות.

בנוסף, אם שני הצדדים פועלים לפי הפרוטוקול אז יש דרישה שבסוף השלב השני המקבלת (R^*) תקבל את הערך שאליו התחייב המתחייב (C^*).

בפרוטוקולים שלנו, נשתמש בסכמת התחייבות שבה ה-*Hiding* חישובי וה-*Binding* סטטיסטי. קיים סוג נוסף של סכמות התחייבות שבה ה-*Hiding* סטטיסטי וה-*Binding* חישובי. צריך להניח שהשולח או המקבלת מוגבלים חישובית ובהתאם לכך הדרישה על אחת התכונות תהיה חישובית (Goldreich, 2001).

פרוטוקול 1: ZKP ל-3 צביעות של גרף

בתחילה, התעסקנו ב-*ZKP* ל-3 צביעות של גרף. נסביר מהי שפת הגרפים ה-3 צביעים ואיך מוגדר ה-*ZKP* עבורה. בהמשך, נתאר את הניסויים שביצענו ואת המסקנות שנובעות מהם.

גרף 3-צביע (Graph 3 Coloring [G3C])

השפה ***G3C*** מורכבת מכל הגרפים הפשוטים שבהם ניתן לצבוע כל קודקוד באחד מתוך שלושה צבעים, כך שלא יהיו 2 קודקודים סמוכים עם אותו צבע.

באופן פורמלי, נגדיר **3-צביעה (*3-coloring*)** של גרף $G = (U, E)$ כפונקציה $\chi: U \to \{1,2,3\}$ כך שלכל $v \in U$, $\chi(v) \in \{1,2,3\}$ הוא "הצבע" של *v*.

צביעה χ היא **צביעה חוקית (*legal*)** אם לכל $e = \{i, j\} \in E$ מתקיים $\chi(i) \neq \chi(j)$

גרף *G* הוא **3-צביע (*G3C*)** אם יש לו 3-צביעה חוקית (*legal 3 coloring*).

הפרוטוקול הראשון שממומש בפרויקט הוא עבור מערכת הוכחה באפס ידיעה (*ZKP*) ל-3 צביעות של גרף. כלומר, נעסוק ב-*ZKP* עבור שייכות לשפה הבאה:

$$L_{3COL} = \{G: G\ is\ 3 - colorable\}$$

ה-ZKP ל-3 צביעות של גרף (ZKP for 3-Colorability [GMW86])

<u>קלט משותף ל-P, V</u>: גרף פשוט $G = (U, E)$

<u>קלט פרטי של *P*:</u> 3-צביעה חוקית χ של G

<u>צעד ראשון של המוכיח *P*:</u>

P בוחר פרמוטציה אקראית $\sigma: \{1,2,3\} \to \{1,2,3\}$, ומגדיר צביעה חדשה לכל קודקוד ע"י הפעלת הפרמוטציה על הצביעה המקורית.

P שולח ל-*V* רצף של $|U|$ התחייבויות על כל אחד מהקודקודים עם הצביעה החדשה ע"י שימוש בסכמת התחייבות. כלומר, *P* מבצע את הפעולה הבאה:

$$\forall i \in U: commit\ to\ \sigma\big(\chi(i)\big)$$

צעד ראשון של המוודאת *V*:

V בוחרת באופן אקראי קשת $e = (i, j) \leftarrow E$ ושולחת אותה ל-*P* (*V* בעצם מבקשת לבדוק את הצבעים של הקודקודים i, j).

צעד שני של המוכיח *P*:

P שולח ל-*V* את הפתיחה של ההתחייבויות עבור הצביעה של קודקודים i, j. כלומר *P* מבצע:

$$Decommit\ \sigma(\chi(i)), \sigma(\chi(j))$$

צעד שני של המוודאת *V*:

V בודקת את הפתיחה של ההתחייבויות, ומקבלת את הטענה אם שתי ההתחייבויות מכילות שני צבעים שונים. כלומר, *V* מבצע:

$$If\ \ \sigma(\chi(i)), \sigma(\chi(j)) \in \{1,2,3\}\ and\ \sigma(\chi(i)) \neq \sigma(\chi(j)) \rightarrow out = accept$$

$$Else \rightarrow out = reject$$

$(Goldreich, Micali, Wigderson, 1986)$

בפרוטוקול הזה, "המטרה" של המוכיח היא להוכיח למוודאת שהגרף 3 צביע. כלומר להוכיח למוודאת שיש לו 3 צביעה חוקית לגרף, מבלי לחשוף את הצביעה עצמה (לכן ההוכחה היא באפס ידיעה).

במערכת הוכחה זו, ההנחה היא שהמוודאת V^* היא *PPT* (מוגבלת חישובית), ושהמוכיח P^* אינו מוגבל חישובית.

אנחנו משתמשים בסכמת התחייבות כך שהמוכיח P^* הוא המתחייב (*Committer*), והמוודאת V^* היא המקבלת (*Receiver*). לכן, נשתמש בסכמת ההתחייבות הבאה:

Computationally hiding statistically binding commitment scheme

כלומר, נדרוש *statistical binding* מכיוון שאנחנו רוצים *binding* גם עבור *committer* שאינו מוגבל חישובית (בפרוטוקול שלנו המוכיח אינו מוגבל חישובית [והוא ה-*committer* בסכמת ההתחייבות]). בנוסף, נוכל להסתפק ב-*computational hiding* כי בפרוטוקול שלנו ה-*receiver* מוגבלת חישובית (המוודאת היא ה-*receiver* והיא *PPT*).

תכנון המימוש של הפרוטוקול

לפני מימוש הפרוטוקול, תכננו את האופן שבו המימוש יתבצע ובחנו חלופות שונות. הנושאים העיקריים שנבחנו:

- <u>התקשורת והאינטראקציה בין שני המשתתפים:</u>

רצינו למצוא דרך נוחה ויעילה לקיים את התקשורת בין שני המשתתפים בפרוטוקול, והתלבטנו בין שימוש בת'רדים עם משאב משותף לבין שימוש בארכיטקטורת שרת-לקוח.

בחרנו לעבוד בארכיטקטורת שרת-לקוח, שבה השרת הוא המוכיח והלקוח הוא המוודאת. בארכיטקטורה זו, התקשורת בין שני המשתתפים יכולה להתבצע משני מחשבים שונים בפשטות והיא מדמה בצורה טובה את העובדה שמדובר בשני משתתפים בלתי תלויים. לעומת זאת, השימוש בת'רדים מחייב גישה למשאב המשותף על מנת שהתקשורת תתבצע כראוי.

כמו כן, השרת הוא בד"כ חזק יותר וזה מתאים לכך שהמוכיח הרמאי הוא לא מוגבל חישובית בפרוטוקול.

- <u>ייצוג הגרף:</u>

עבור הייצוג של הגרף התלבטנו בין שימוש במטריצת שכנויות לבין רשימת שכנויות. ראינו כי לכל אחד מהייצוגים יש יתרונות וחסרונות שונים, והכדאיות של כל ייצוג עשויה להיות תלויה באפליקציה הספציפית שבה משתמשים בפרוטוקול. המימוש שלנו לא מיועד לאפליקציה מסוימת, ולכן שני הייצוגים היו הגיוניים לבחירה. החלטנו לעבוד עם מטריצת שכנויות מכיוון שהייצוג הזה יהיה שימושי גם בפרוטוקול הבא.

- <u>ייצוג הצביעה של הגרף:</u>

החלטנו לייצג את הצביעה ע"י מערך בגודל של מספר הקודקודים. המערך מכיל בכל תא את אחד מהערכים 1/2/3 (כל ערך מייצג צבע שונה), כך שהצביעה של כל קודקוד i שמורה במערך במקום ה-i.

- <u>סכמת התחייבות:</u>

לצורך מימוש הפרוטוקול, נדרשנו להחליט באיזו סכמת התחייבות לעבוד. הסכמה צריכה להיות יעילה ולקיים את הדרישות הספציפיות של התכונות (*Computationally hiding statistically binding*).

לאחר חיפוש ובחינה של מספר סכמות התחייבות, לא מצאנו מימוש שהולם את התכונות שאנחנו צריכים, פועל באופן יעיל וגם ניתן לשימוש בפייתון. קראנו על סכמה שמקיימת את התכונות הנדרשות שנקראת – *ElGamal*, והחלטנו לממש אותה בעצמנו.

<u>הסכמה מוגדרת באופן הבא:</u>

[תיאור הסכמה נלקח מ- (Fernàndez-València, 2021)]

קלט: G - חבורה ציקלית, q – סדר החבורה, g - איבר יוצר של החבורה, y – המפתח הפומבי.

שלב ההתחייבות (*Commit phase*): על מנת להתחייב על ערך $m \in G$, המתחייב מגריל בהתפלגות אקראית ערך $r \in Z_q$. ההתחייבות תהיה: $c = (g^r, my^r)$.

שלב הפתיחה (*Decommit phase*): על מנת לפתוח התחייבות c, המתחייב חושף למקבלת את הזוג (m, r), כאשר m מייצג את הערך שעליו בוצעה ההתחייבות. המקבלת בודקת אם מתקיים $c = (g^r, my^r)$. במידה וכן, הפלט שלה יהיה "קבלה" (ההתחייבות תקינה והערך שנחשף הוא m). אחרת, הפלט שלה יהיה "דחייה".

<u>מימוש הפרוטוקול</u>

מימשנו את הפרוטוקול בשפת פייתון, והעלינו אותו לגיט **[נספח 1: קישור לגיט של הפרויקט ב-GitHub]**.

המימוש מכיל את קבצי הפייתון עם השמות הבאים:

Prover, Verifier, CommonGraph, PrivateColoring, ElGamalCommitmentScheme.

בקובץ <u>ElGamalCommitmentScheme.py</u> **[נספח 2: ElGamalCommitmentScheme.py]** מימשנו את סכמת ההתחייבות של ElGamal, נסביר את המתודות המרכזיות שהוא מכיל:

- **generateKeys** – מתודה שמקבלת את היוצר של החבורה ואת הסדר שלה, ויוצרת את המפתח הפרטי והפומבי.
- **commit** – מתודה שמקבלת את הערכים הידועים (q, g, y) וערך מסוים - value, ומתחייבת על הערך לפי סכמת ההתחייבות של ElGamal. המתודה מחזירה את ההתחייבות - C ואת הערך הרנדומלי שהוגרל (הערך שישמש לפתיחה) - r.
- **Verify** – מתודה בוליאנית שמקבלת את הערכים הידועים, מחרוזת dec (value, r), ואת ההתחייבות על הערך value, ובודקת את הפתיחה לפי סכמת ההתחייבות של

ElGamal. היא מחזירה True אם הפתיחה נכונה בהתאם להתחייבות, ו-False אחרת.

בקובץ PrivateColoring.py **[נספח 3: PrivateColoring.py]** שמורה רשימה עם הצביעה של הגרף שידועה למוכיח (הקלט הפרטי של המוכיח). בקובץ הזה יש מתודה בשם **getPrivateColoring** שמחזירה את הצביעה הידועה.

בקובץ CommonGraph.py **[נספח 4: CommonGraph.py]** מיוצר הגרף שעליו מפעילים את הפרוטוקול (הקלט המשותף של המוכיח והמוודאת). בקובץ הזה יש מתודה בשם **getCommonGraph** שמחזירה את הגרף, והיא משמשת את המוכיח והמוודאת.

כמו כן, בקובץ יש מספר מתודות שמשמשות ליצירת גרפים שונים ולייצוגם כמטריצת שכנויות, כמו: createGraphByAdjMatrix (מתודה שמקבלת מהמשתמש רצף של קשתות ויוצרת מטריצת שכנויות שמייצגת את הגרף) ו-loadGraphAsAdjMatrix (מתודה שמקבלת קובץ טקסט שמכיל קשתות של גרף, ויוצרת ממנו מטריצת שכנויות שמייצגת את הגרף).

בקובץ Prover.py **[נספח 5: Prover.py]** מתבצע החלק של המוכיח בפרוטוקול. נסביר את המתודות המרכזיות שהוא מכיל:

- **start_server** – מתודה שמייצרת את התקשורת בין המוכיח למוודאת, וקוראת למתודה **handle_client** שבה מתבצע הצד של המוכיח בפרוטוקול.
- **firstStepOfProver** – מתודה שמקבלת את ה-socket, את הערכים הידועים ואת הצביעה של הגרף, ומבצעת את הצעד הראשון של המוכיח לפי הפרוטוקול (הפעלת פרמוטציה על הצמתים והתחייבות על הצביעה החדשה).
- **secondStepOfProver** – מתודה שמקבלת את ה-socket, את הפרמוטציה, את הערכים הרנדומליים שמשמשים לפתיחה ואת הקשת שנבחרה (הקשת מיוצגת ע"י זוג צמתים). המתודה מבצעת את הצעד השני של המוכיח לפי הפרוטוקול (שליחת פתיחת התחייבויות עבור הצבעים של 2 הצמתים שמרכיבים את הקשת שנבחרה).

בקובץ Verifier.py **[נספח 6: Verifier.py]** מתבצע החלק של המוודאת בפרוטוקול. נסביר את המתודות המרכזיות שהוא מכיל:

- **start_client** – מתודה שמייצרת את החיבור של הלקוח לשרת עבור התקשורת בין המוכיח למוודאת, ולאחר מכן מבצעת את הצד של המוודאת בפרוטוקול.
- **firstStepOfVerifier** – מתודה שמקבלת את ה-socket ואת הגרף המשותף, ומבצעת את הצעד הראשון של המוודאת לפי הפרוטוקול (הגרלת קשת באקראי ושליחתה למוכיח).
- **secondStepOfVerifier** – מתודה שמקבלת את ה-socket, את הקשת הנבחרת, את הערכים הידועים, את הפתיחה של צבעי הקודקודים שמרכיבים את הקשת הנבחרת ואת מערך ההתחייבויות של הצבעים. המתודה מבצעת את הצעד השני של המוודאת לפי הפרוטוקול (בדיקה שפתיחת התחייבויות של צבעי הקודקודים שמרכיבים את הקשת הנבחרת היא תקינה, והחלטה על קבלת הטענה או דחייתה בהתאם לפרוטוקול).

לצורך המימוש נעזרנו במספר ספריות עיקריות: הספריות socket ו-threading שימשו למימוש התקשורת בין המוכיח ומוודאת, הספרייה random שימשה לבחירת קשת אקראית ולביצוע פעולות שונות במימוש סכמת ההתחייבות והספרייה numpy שימשה ליצירת פרמוטציה אקראית על הצבעים.

לצורך הרצת הפרוטוקול, מריצים את הקובץ Prover.py ולאחריו את הקובץ Verifier.py. המוכיח והמוודאת יוצרים את התקשורת ביניהם, ולאחר מכן מתבצעת הריצה לפי שלבי הפרוטוקול.

האתגרים במימוש הפרוטוקול

במהלך מימוש הפרוטוקול נתקלנו במספר אתגרים. בכל פעם שנתקלנו בקושי בנושא מסוים, קראנו עליו במספר מקורות והחלטנו על הדרך הנכונה ביותר להתמודד איתו. נתאר את האתגרים המרכזיים במימוש הפרוטוקול ואת ההתמודדות איתם:

- במימוש של סכמת ההתחייבות, יש צורך לקבוע את הגודל של פרמטר הבטיחות. צריך לבחור ערך מתאים כך שמצד אחד תתקבל הבטיחות הרצויה ומצד שני זמן הריצה יהיה סביר. לצורך כך, חיפשנו מקור אמין להבנת הפרמטרים המקובלים בתחום. לאחר קריאת מקורות שונים, מצאנו מאמר של *NIST* עם הסבר על הערכים המקובלים של פרמטר הבטיחות עבור פרוטוקול מסוג זה (Barker, 2016). בהתאם לכך, החלטנו לבחור את הגודל של פרמטר הבטיחות להיות ערך אקראי בין 2^{224} ל- 2^{256}.

- בתקשורת בין המוכיח למוודאת מועברים רק ערכים מסוג מחרוזת. אנחנו מעבירים ערכים שונים שחלקם מספרים וכן אמורים להישמר במבני נתונים שונים כמו מערכים. לצורך כך, כתבנו מתודות וקטעי קוד שמעבדים את הנתונים שמועברים או מתקבלים, וממירים אותם לטיפוס הנתונים הרצוי.

- לצורך דגימת קשת אקראית, חשבנו בתחילה להגריל שני צמתים באקראי ולבדוק אם יש ביניהם קשת, ובמידה ולא לחזור על הפעולה עד שתתקבל קשת מהגרף. כשניתחנו את ההסתברות לבחירה, ראינו כי על אף ששני הצמתים מוגרלים באקראי, ההתפלגות המתקבלת על הקשתות איננה אקראית.
לכן, לצורך הבחירה של קשת אקראית החלטנו לייצג את הקשתות ע"י מבנה נתונים של רשימה ולבחור ממנה קשת באקראי.

ניסויים עבור ה-ZKP ל-3 צביעות של גרף

ניסויים לבדיקת נכונות המימוש

בתחילה, רצינו לבדוק את נכונות המימוש. כלומר, רצינו לראות שהתרגום של הפרוטוקול לתוכנית שיצרנו הוא נכון ולא מכיל שגיאות.

לצורך כך, יצרנו מספר גרפים שחלקם 3-צביעים וחלקם לא, הגדרנו להם צביעות שחלקן 3 צביעות חוקיות וחלקן לא, ועבור כל אחד מהגרפים ומהצביעות שמתאימים להם הרצנו את התוכנית מספר פעמים וראינו שהתוצאות אכן נכונות.

נראה דוגמה לאחד מהניסויים האלה:

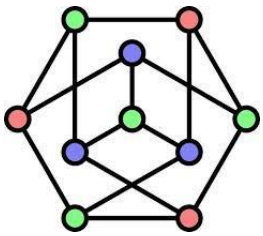
הגרף המשותף

```
# A 3-coloring graph with 10 vertices and 15 edges
G2 = [[0, 1, 0, 0, 0, 1, 0, 0, 0, 1], [1, 0, 1, 0, 0, 0, 0, 0, 1, 0],
[0, 1, 0, 1, 0, 0, 1, 0, 0, 0], [0, 0, 1, 0, 1, 0, 0, 0, 0, 1], [0,
0, 0, 1, 0, 1, 0, 0, 1, 0], [1, 0, 0, 0, 1, 0, 1, 0, 0, 0], [0, 0, 1,
0, 0, 1, 0, 1, 0, 0], [0, 0, 0, 0, 0, 0, 1, 0, 1, 1], [0, 1, 0, 0, 1,
0, 0, 1, 0, 0], [1, 0, 0, 1, 0, 0, 0, 1, 0, 0]]
```

הצביעה של הגרף (הקלט הפרטי של המוכיח)

```
coloring2 = [2, 1, 2, 1, 2, 1, 3, 2, 3, 3]  # the coloring for the
graph G2
```

G2 מכיל את הייצוג של הגרף הבא כמטריצת שכנויות ו-*coloring2* מכיל את הייצוג של הצביעה שלו (*1, 2, 3* מייצגים את הצבעים אדום, ירוק, כחול בהתאמה):



הפלט של ההרצה

```
Verifier:

Perform the first step of the verifier
The chosen edge:  [2, 6]
Perform the second step of the verifier

Accept the claim - the graph is a 3-coloring!

The interaction with the prover has finished
```

```
Prover:

Perform the first step of the prover
The verifier chose the edge:  [2, 6]
Perform the second step of the prover

The verifier accepted the claim!
```

ניתן לראות שכל אחד מהמשתתפים בפרוטוקול מבצע את השלבים שלו. הקשת שנבחרה בהרצה זו היא הקשת [*2, 6*] שאכן קיימת בגרף. בסופו של דבר, המוודאת קיבלה את הטענה והחליטה שהגרף הוא 3 צביע.

ניסויים לבדיקת יעילות המימוש ביחס לפרמטרים שונים

רצינו לבדוק מדדי יעילות שונים של התוכנית. המדדים שהחלטנו לבדוק הם זמן ריצה, צריכת זיכרון ותקשורת. המדדים האלה הם המדדים העיקריים שמושפעים מהקלט ומגדירים את היעילות והביצועים של הקוד.
נרצה לבחון את ההשפעה של מספר הצמתים ואת ההשפעה של מספר הקשתות על כל אחד מהמדדים. כך נוכל להבין מה משפיע בצורה משמעותית על ערכי המדדים השונים בפרוטוקול וכן להסיק עבור איזה מדדים ועבור איזה קלטים הפרוטוקול יעיל.

לצורך כך, חיפשנו גרפים גדולים מאוד שנוכל לבחון בעזרתם את הביצועים של הקוד.
לאחר מעבר על מקורות מידע רבים של גרפים, מצאנו אתר בשם *SNAP* שמכיל גרפים גדולים שנלקחו ממחקר של ניתוח רשתות חברתיות ומידע גדולות באוניברסיטת סטנפורד (Leskovec, n.d.).

במקור המידע הזה מצאנו שני גרפים מספיק גדולים ששימשו אותנו לצורך הניסויים:

1. ***Facebook*** – הגרף ששמור ומיוצג בקובץ *facebook_combined.txt*.
   בגרף זה יש 4,039 צמתים ו-88,234 קשתות, והוא מתאר רשתות חברתיות של משתמשים בפייסבוק. השתמשנו בגרף הזה עבור בדיקת ההשפעה של מספר הקשתות על המדדים השונים.

2. ***p2p-Gnutella04*** – הגרף ששמור ומיוצג בקובץ *p2p-Gnutella04.txt*.
   בגרף זה יש 10,879 צמתים ו-39,994 קשתות, והוא מתאר רשת שיתוף הקבצים בין משתמשים. השתמשנו בגרף הזה עבור בדיקת ההשפעה של מספר הצמתים על המדדים השונים.

החלטנו לעבוד עם הגרפים האלה מכיוון שהם מספיק גדולים כדי להציג לנו את ההשפעה של המדדים השונים. יתר על כן, ראינו שיש עליהם מספיק מידע כדי להבין מה הם מתארים, וכן שהם מופיעים במקור מידע אמין שיכול לשמש לניסויי מחקר.
נדגיש כי הגרפים האלה משמשים רק עבור בדיקת מדדי היעילות ולא ידוע לנו אם הם 3-צביעים.

לצורך בדיקת ההשפעה של מספר הקשתות, יצרנו בקובץ *CommonGraph.py* מתודה בשם *createCheckingGraphByEdges*. המתודה הזאת מקבלת את הגישה לקובץ הגרף המקורי, את האינדקס שבו מתחיל להופיע רצף הקשתות בקובץ ואת מספר הקשתות הדרוש. היא יוצרת תת-גרף של הגרף המקורי עם מספר קשתות שזהה למספר הקשתות הדרוש, ושומרת את תת-הגרף בקובץ חדש עם שם מתאים.

באופן דומה, לצורך בדיקת ההשפעה של מספר הצמתים יצרנו בקובץ *CommonGraph.py* מתודה בשם *createCheckingGraphByNodes*. המתודה הזאת מקבלת את הגישה לקובץ הגרף המקורי, את האינדקס שבו מתחילים להופיע רצף הקשתות בקובץ ואת מספר הצמתים הדרוש. היא יוצרת תת-גרף של הגרף המקורי עם מספר צמתים שזהה למספר הצמתים הדרוש, ושומרת את תת-הגרף בקובץ חדש עם שם מתאים.

כמו כן, יצרנו מתודה בשם *loadCheckingGraphAsAdjMatrix* שמקבלת את הגישה לקובץ של תת גרף שנוצר, ושומרת אותו בייצוג של מטריצת שכנויות.

עבור בדיקת ההשפעה של מספר הקשתות, יצרנו מספר תתי-גרפים של גרף ה-*Facebook* עם מספר צמתים זהה ועם מספר קשתות שגדל בקפיצות של 8000.
בנוסף, לבדיקת ההשפעה של מספר הצמתים, יצרנו מספר תתי-גרפים של גרף ה-*Gnutella* עם מספר צמתים שגדל (ובהתאם לכך גם מספר הקשתות גדל) בקפיצות של 1000.

עבור הניסויים חיפשנו ספריות שמשמשות למדידת המדדים שמעניינים אותנו. לאחר חיפוש נרחב, מצאנו ספריות מוכרות שמבצעות את המדידות בצורה טובה. הספרייה *timeit* משמשת למדידת זמני הריצה של המוכיח והמוודאת, הספרייה *psutil* משמשת למדידת צריכת הזיכרון של המוכיח והמוודאת והספרייה *sys* משמשת למדידת מספר הביטים שעוברים בתקשורת.

הוספנו בקוד של המוכיח והמוודאת פקודות למדידת זמני הריצה וצריכת הזיכרון של כל אחד בנפרד. כמו כן, בצד המוכיח הוספנו פקודות למדידת מספר הביטים שעוברים בתקשורת בין המוכיח למוודאת (התקשורת היא מדד שמשותף לשניהם).
עבור מדידת צריכת הזיכרון, מדדנו את כל הזיכרון שבשימוש בהרצת התוכנית ע"י כל אחד מהמשתתפים בנפרד. לעומת זאת, עבור מדידת מספר הביטים שעוברים בתקשורת, מדדנו את גודל הזיכרון של האובייקטים שנשלחים בתקשורת.

לכל תת גרף, הרצנו את הפרוטוקול עבור כל אחת מהמדידות וקיבלנו את ערכי המדדים שמתאימים לו. הכנסנו את הערכים לטבלאות באקסל, ובעזרתם יצרנו גרפים מתאימים עם

קווי מגמה ומשוואות. הגרפים ישמשו להבנת הסיבוכיות האסימפטוטית של ערכי המדדים כתלות במספר הצמתים ובמספר הקשתות.

תוצאות הניסויים – השפעת מספר הצמתים:

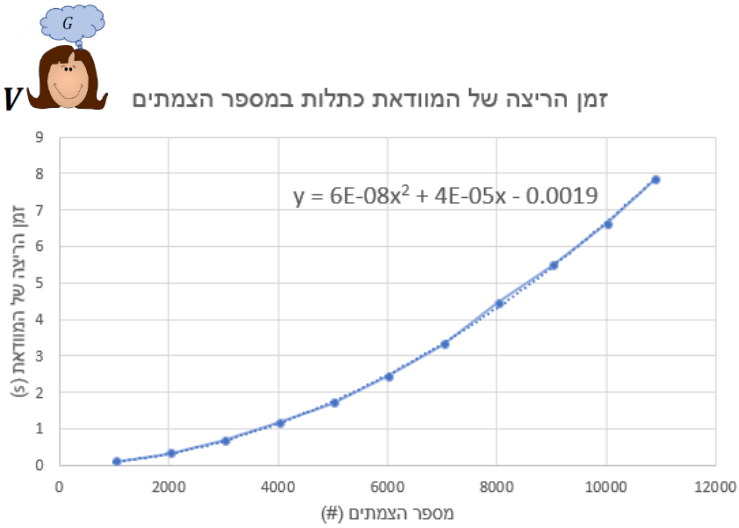
לאחר ביצוע המדידות לבדיקת ההשפעה של מספר הצמתים, ריכזנו את ערכי המדדים שהתקבלו בטבלה הבאה:

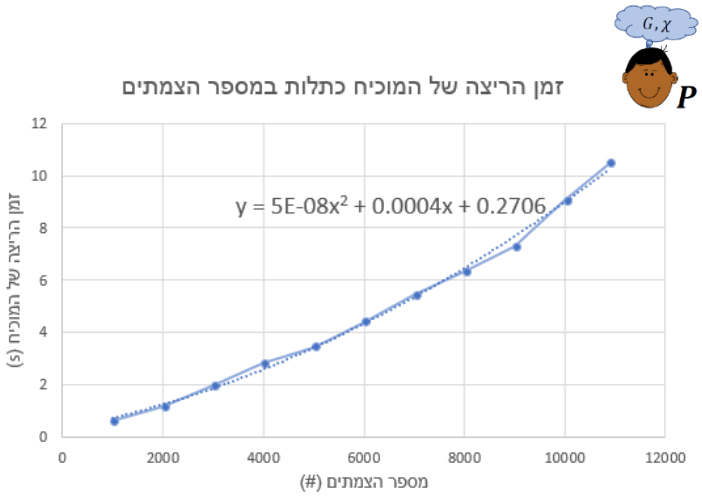
מספר הצמתים (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
1000	0.598	0.109	0.153	10.102	10.637
2000	1.15	0.325	0.304	38.426	35.273
3000	1.95	0.68	0.458	79.598	81.781
4000	2.802	1.164	0.61	133.469	135.57
5000	3.445	1.699	0.761	210.453	212.195
6000	4.388	2.428	0.895	282.02	286.445
7000	5.421	3.307	1.067	407.121	409.16
8000	6.322	4.436	1.219	527.25	529.551
9000	7.273	5.471	1.362	664.008	667.191
10000	9.029	6.613	1.527	812.145	815.59
10879	10.486	7.815	1.657	918.086	920.531

הגרף המלא

זמן ריצה

מהמדידות של זמני הריצה קיבלנו את הגרפים הבאים:





ניתן לראות שזמן הריצה של המוכיח והמוודאת עולים ככל שמספר הצמתים עולה.

נשים לב שזמני הריצה של התוכנית הם מהירים יחסית, ושעבור גרפים עם עשרות אלפי צמתים המוכיח והמוודאת מסיימים את ריצתם בשניות בודדות.

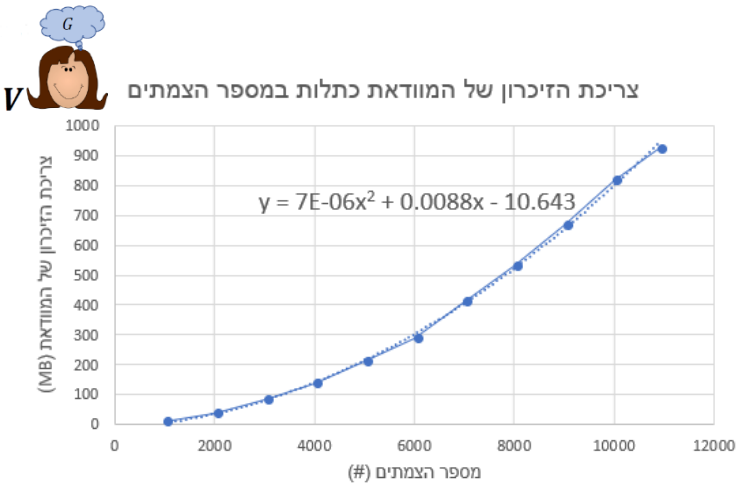
כמו כן, מהגרף עולה כי זמן הריצה של המוכיח הוא יותר גדול מזה של המוודאת. דבר זה קורה בעיקר מכיוון שהמוכיח הוא זה שמחשב את ההתחייבויות על הצביעה, וחישוב ההתחייבויות לוקח זמן רב יחסית.

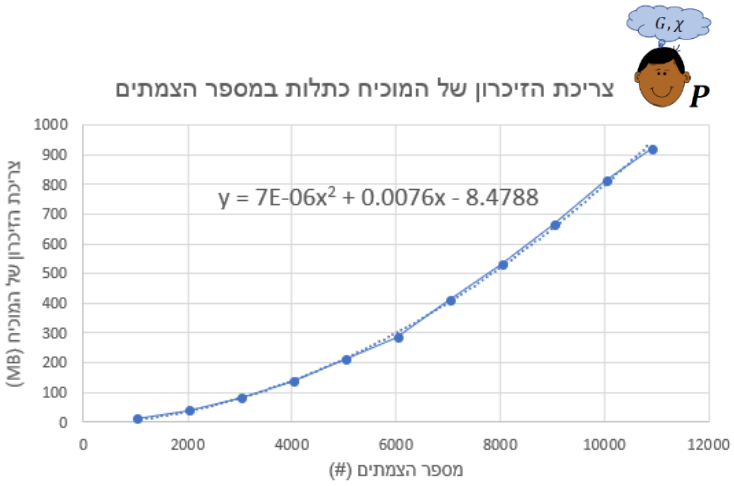
הגרף של המוכיח הוא בקירוב פולינומי מסדר 2. הסיבה לכך היא שהקריאה והייצוג של הגרף במטריצת שכנויות לוקחים זמן של $O(|E|+|U|^2)$. שאר הפעולות (כמו הפעלת הפרמוטציה על הצביעה וההתחייבויות עליה) מתבצעות לכל היותר בזמן לינארי $O(|U|)$.

הגרף של המוודאת הוא גם כן בקירוב פולינומי מסדר 2. הסיבה לכך היא שגם כאן יש השפעה זהה לקריאה והייצוג של הגרף במטריצת שכנויות, וכן המוודאת בוחרת קשת אקראית שלוקחת זמן ריצה של $O(|U|^2)$.

<u>צריכת הזיכרון</u>

מהמדידות של צריכת הזיכרון קיבלנו את הגרפים הבאים:





ניתן לראות שצריכת הזיכרון של המוכיח והמוודאת עולות ככל שמספר הצמתים עולה.

נשים לב שצריכת הזיכרון של המוכיח והמוודאת הן קרובות מאוד בכל ההרצות. שניהם שומרים כמעט את אותם הנתונים (למשל שניהם שומרים את הגרף המשותף, את ההתחייבויות על הצביעה של כל הצמתים ואת הקשת הנבחרת), ולכן הם משתמשים בכמות זיכרון דומה.

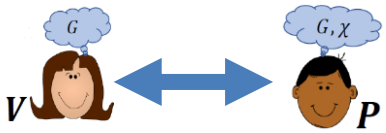
הגרף של המוכיח הוא בקירוב פולינומי מסדר 2, וזאת מכיוון שהייצוג של הגרף במטריצת שכנויות דורש זיכרון של $O(|U|^2)$. שאר האובייקטים (כמו מבני הנתונים ששומרים את הצביעה ואת ההתחייבויות) דורשים לכל היותר גודל זיכרון לינארי של $O(|U|)$.

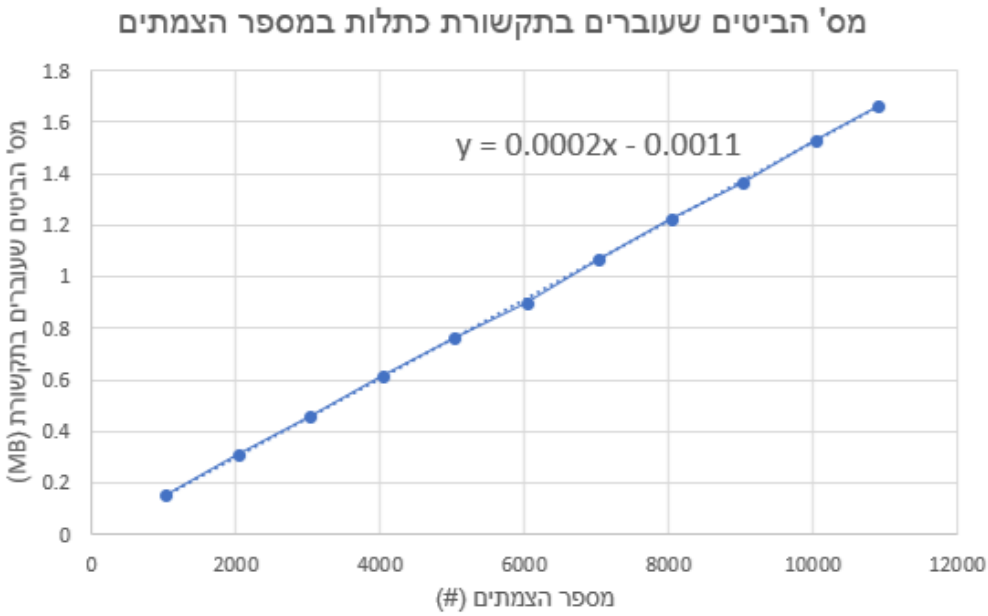
הגרף של המוודאת הוא גם כן בקירוב פולינומי מסדר 2. הסיבה לכך היא שגם כאן יש השפעה זהה לייצוג של הגרף במטריצת שכנויות. שאר האובייקטים (כמו מבני הנתונים

ששומרים את ההתחייבויות ואת רשימת הקשתות) דורשים לכל היותר גודל זיכרון לינארי של $O(|U|)$ או $O(|E|)$.

<u>תקשורת</u>

מהמדידות של מס' הביטים שעוברים בתקשורת קיבלנו את הגרף הבא:





ניתן לראות שמספר הביטים שעוברים בתקשורת בין המוכיח למוודאת עולה ככל שמספר הצמתים עולה.

נשים לב כי מספר הביטים שעוברים בתקשורת בין המוכיח למוודאת היא לינארית ביחס למספר הצמתים - $O(|U|)$. זאת מכיוון שבתקשורת מועברות ההתחייבויות על הצבעים של כל אחד מהצמתים.

שאר הנתונים שמועברים בתקשורת (קשת נבחרת, חשיפה של שתי התחייבויות והתוצאה [ההחלטה של המוודאת]) דורשים זיכרון קבוע - $O(1)$.

תוצאות הניסויים – השפעת מספר הקשתות:

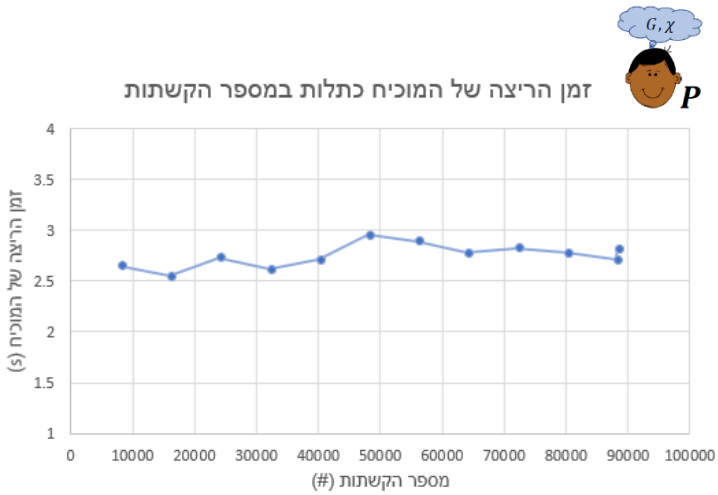
לאחר ביצוע המדידות לבדיקת ההשפעה של מספר הקשתות, ריכזנו את ערכי המדדים שהתקבלו בטבלה הבאה:

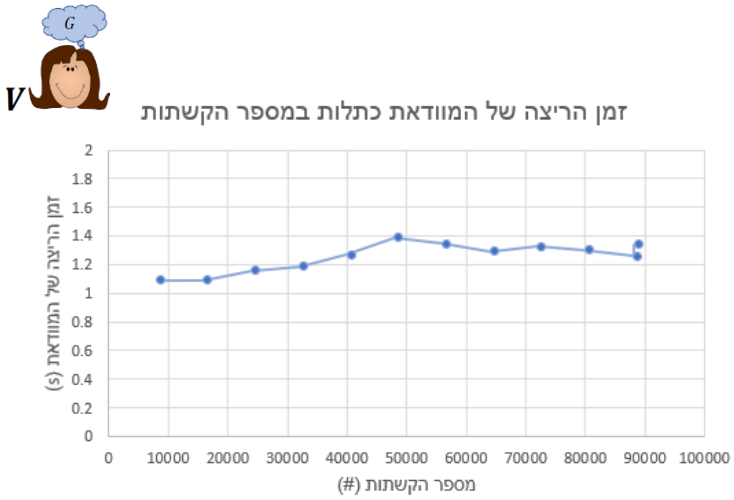
מספר הקשתות (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
8000	2.652	1.092	0.615	134.867	136.383
16000	2.554	1.091	0.617	135.277	137.289
24000	2.734	1.157	0.615	135.113	136.738
32000	2.621	1.189	0.612	135.16	137.074
40000	2.709	1.267	0.609	134.996	137.395
48000	2.958	1.391	0.616	135.422	137.387
56000	2.897	1.347	0.615	135.641	136.629
64000	2.781	1.292	0.611	135.086	137.16
72000	2.827	1.328	0.616	135.051	137.027
80000	2.782	1.304	0.615	135.332	137.07
88000	2.709	1.261	0.616	134.965	137.238
88234	2.82	1.342	0.614	134.902	137.696

הגרף המלא

זמן ריצה

מהמדידות של זמני הריצה קיבלנו את הגרפים הבאים:





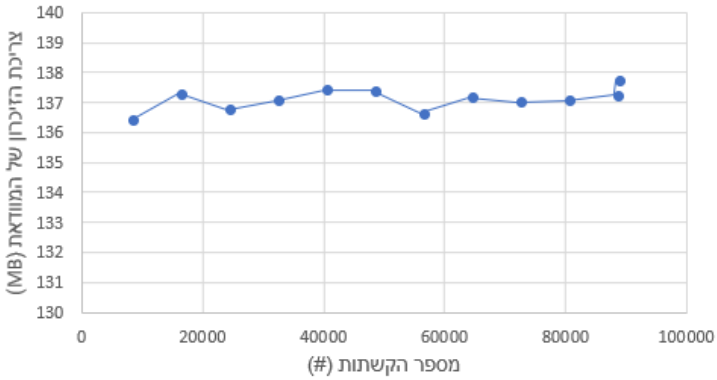
ניתן לראות שזמני הריצה של המוכיח והמוודאת קבועים בקירוב ככל שמספר הקשתות עולה (מספר הצמתים קבוע).

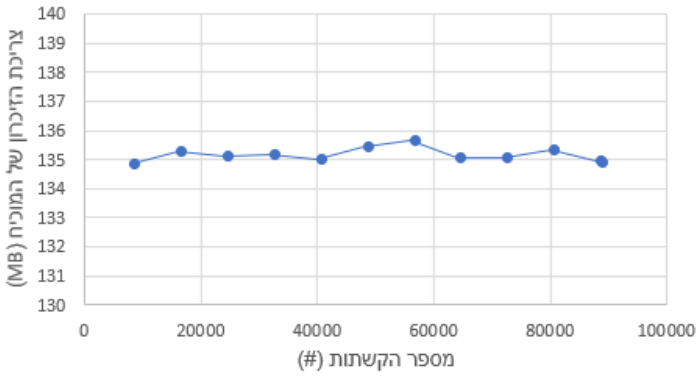
זמני הריצה של הפעולות שמתבצעות בריצת המוכיח והמוודאת (כמו: קריאת הגרף, הפעלת הפרמוטציה על הצביעה וחישוב ההתחייבויות על הצבעים) תלויים בעיקר במספר הצמתים ולא במספר הקשתות. כלומר, זמני הריצה של המוכיח והמוודאת אינם מושפעים בצורה משמעותית מכמות הקשתות, ולכן הגרפים של המוכיח ושל המוודאת הם קבועים בקירוב.

כפי שראינו קודם, גם כאן זמן הריצה של המוכיח הוא יותר גדול מזה של המוודאת, בעיקר בגלל שבו מחושבות ההתחייבויות על הצביעה (פעולה שלוקחת זמן רב יחסית).

צריכת הזיכרון

מהמדידות של צריכת הזיכרון קיבלנו את הגרפים הבאים:





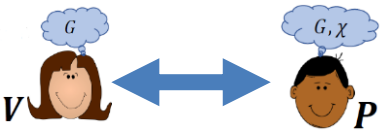
ניתן לראות שצריכת הזיכרון של המוכיח ושל המוודאת קבועות בקירוב ככל שמספר הקשתות עולה.

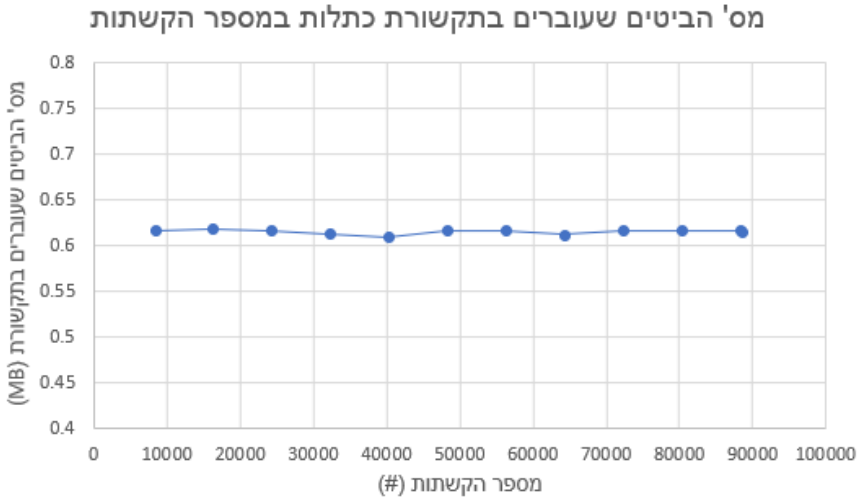
צריכת הזיכרון במהלך הריצה של המוכיח ושל המוודאת (כמו: הזיכרון עבור שמירת הייצוג של הגרף, הפרמוטציה על הצביעה וההתחייבויות על הצבעים) תלויה בעיקר במספר הצמתים ולא במספר הקשתות. כלומר, צריכת הזיכרון של המוכיח והמוודאת אינן מושפעות בצורה משמעותית מכמות הקשתות, ולכן הגרפים של המוכיח ושל המוודאת הם קבועים בקירוב.

כפי שראינו קודם, גם כאן צריכת הזיכרון של המוכיח קרובה מאוד לזו של המוודאת בכל ההרצות, מכיוון ששניהם שומרים כמעט את אותם הנתונים.

<u>תקשורת</u>

מהמדידות של מס' הביטים שעוברים בתקשורת קיבלנו את הגרף הבא:





ניתן לראות שמספר הביטים שעוברים בתקשורת בין המוכיח למוודאת הוא קבוע בקירוב ככל שמספר הקשתות עולה.

כפי שראינו קודם, מספר הביטים שעוברים בתקשורת במהלך הריצה של המוכיח והמוודאת תלוי רק במספר הצמתים ולא במספר הקשתות (בתקשורת מועברים ההתחייבויות על הצבעים של כל אחד מהצמתים, הקשת הנבחרת, החשיפה של שתי ההתחייבויות והתוצאה [ההחלטה של המוודאת], וכל אלה אינם תלויים במספר הקשתות).
כלומר, מספר הביטים שעוברים בתקשורת בין המוכיח למוודאת אינה מושפעת מכמות הקשתות, ולכן הגרף הוא קבוע בקירוב.

תיארנו את המימוש, את הניסויים ואת התוצאות שהתקבלו עבור ה-ZKP ל-3 צביעות של גרף. מהתוצאות שהתקבלו, הסקנו מסקנות בנוגע לנכונות הקוד ולהשפעה של מספר הצמתים ומספר הקשתות על הערכים של המדדים השונים.
כעת, נתאר בעיית NP נוספת ונראה את ה-ZKP עבורה.

פרוטוקול 2: ZKP למעגל המילטוני בגרף

בהמשך, התעסקנו ב-*ZKP* למעגל המילטוני בגרף. נסביר מהי שפת הגרפים המכוונים שיש להם מעגל המילטוני ואיך מוגדר ה-*ZKP* עבורה. לאחר מכן, נתאר את הניסויים שהרצנו ואת המסקנות שנובעות מהם.

מעגל המילטוני בגרף (Hamiltonian cycle [HC])

השפה ***HC*** מורכבת מכל הגרפים המכוונים שבהם קיים מעגל המיליטוני.

מעגל המילטוני (*HC*) בגרף מכוון $G = (U, E)$ הוא מעגל כך שמתקיים:

- המעגל עובר בכל צומת פעם אחת בדיוק (חוץ מצומת ההתחלה שבו מבקרים פעמיים).
- המעגל מתחיל ומסתיים באותו הצומת (כלומר המסלול הוא באמת מעגל).

ב-*ZKP* עבור שפה זו, הגרף מיוצג ע"י **מטריצת שכנויות**:
גרף $G = (U, E)$ מיוצג ע"י מטריצה בינארית $|U| \times |U|$ כך שלכל i, j הערך של המטריצה במקום ה-(i, j) הוא 1 אם"ם בגרף יש קשת בין צומת i לצומת j [$(i, j) \in E$].

בפרויקט שלנו אנחנו עוסקים בפרוטוקול של מערכת הוכחה באפס ידיעה לגרפים מכוונים עם מעגל המילטוני. כלומר, נעסוק ב-*ZKP* עבור שייכות לשפה הבאה:

$$L_{HC} = \{G: G\ has\ a\ Hamiltonian\ cycle\}$$

ה-ZKP למעגל המילטוני בגרף (*ZKP for Hamiltonian cycle [Blu86]*)

<u>קלט משותף ל-P, V</u>: גרף מכוון $G = (U, E)$ *שמיוצג ע"י מטריצת שכנויות* M

<u>קלט פרטי של P</u>: מעגל המילטוני C בגרף G

<u>צעד ראשון של המוכיח P:</u>
P בוחר פרמוטציה אקראית $\sigma: |U| \to |U|$, מפעיל אותה על G ומקבל מטריצת שכנויות חדשה M' (מטריצה השכנויות של $\sigma(G)$).

P שולח ל-V רצף של $|U|^2$ התחייבויות על כל הכניסות של מטריצת השכנויות, על כל כניסה בנפרד ($\forall\, i,j \in U:\ commit\ to\ M'_{ij}$).

<u>צעד ראשון של המוודאת V:</u>

V בוחרת באופן אקראי ביט $b \leftarrow \{0,1\}$ ושולחת אותו ל-P

<u>צעד שני של המוכיח P:</u>

אם $b = 0$:

P חושף את התחייבויות על M' ואת הפרמוטציה σ [$dec = (\sigma + decommit\ of\ M')$].

אם $b = 1$:

P חושף את ההתחייבויות של הכניסות שמתאימות לקשתות של המעגל בגרף החדש [$dec = decommit\ of\ edges\ of\ \sigma(C)$].

<u>צעד שני של המוודאת V:</u>

V בודקת את הפתיחה של ההתחייבויות, ומקבלת את הטענה אם פתיחת ההתחייבויות היא נכונה ($decommit\ valid$), וגם מתקיימת הבדיקה הבאה:

אם $b = 0$: המוודאת בודקת שהגרף החדש שהתקבל מהמוכיח איזומורפי לגרף המקורי ($dec\ consistent\ with\ G$).

אם $b = 1$: המוודאת בודקת שהמעגל שהתקבל מהמוכיח הוא מעגל המילטוני ($dec\ is\ Hamilton\ cycle$).

אחרת, V דוחה את הטענה.

$(Blum, 1986)$

בפרוטוקול הזה, "המטרה" של המוכיח היא להוכיח למוודאת שהגרף הוא בעל מעגל המילטוני. כלומר, להוכיח למוודאת שיש מעגל המילטוני בגרף, מבלי לחשוף את המעגל עצמו (לכן ההוכחה היא באפס ידיעה).

גם במערכת הוכחה זו, ההנחה היא שהמוודאת V^* היא *PPT* (מוגבלת חישובית), ושהמוכיח P^* אינו מוגבל חישובית, ולכן נשתמש בסכמת התחייבות בעלת אותן הדרישות בדיוק (*Computationally hiding statistically binding commitment scheme*).

התכנון והאתגרים בהתאמת המימוש לפרוטוקול החדש

קיים דמיון רב בין המימושים של שני הפרוטוקולים, שבא לידי ביטוי בתקשורת בין המוכיח והמוודאת (ארכיטקטורת שרת-לקוח), בייצוג של הגרף (מטריצת שכנויות) ובסכמת ההתחייבות (המימוש של סכמת ElGamal).

עם זאת, נדרשנו לבצע מספר התאמות עבור המימוש של הפרוטוקול החדש. נתאר את האתגרים העיקריים בהתאמה של המימוש:

- ייצוג המעגל ההמילטוני:
  החלטנו לייצג את המעגל ההמילטוני ע"י מערך שמכיל רצף של קודקודים שמתארים את המעגל לפי הסדר.

- יצירת גרף חדש בהתאם לפרמוטציה שמופעלת על הקודקודים:
  בשלב הראשון של המוכיח, הוא צריך להגריל פרמוטציה על הקודקודים וליצור מטריצת שכנויות חדשה בהתאם.
  לכל מקום i,j שהכיל את הערך 1 (שמייצג קשת) בגרף המקורי, שמנו את הערך 1 במקום המתאים במטריצה החדשה לפי הפרמוטציה על הצמתים i ו-j.

- שמירת ההתחייבויות של מטריצת השכנויות:
  בפרוטוקול מתחייבים על כל אחת מהכניסות של מטריצת השכנויות (לעומת הפרוטוקול הקודם, שבו התחייבנו על רצף של צבעים שהיו שמורים מראש במערך חד מימדי), וצריך לשמור ולהעביר את ההתחייבויות מהמוכיח למוודאת.
  לצורך כך, שמרנו את ההתחייבויות במערך חד מימדי ארוך שמכיל את הערכים של הכניסות שורה אחר שורה (כך נשמרו ההתחייבויות לפי הסדר של הכניסות במטריצה שורה אחר שורה).

- אקראיות שמשפיעה על פעולות המוכיח והמוודאת:
  בשלב השני של המוכיח והמוודאת, האופן שבו הם פועלים מושפע מהביט שהוגרל ע"י המוודאת. לכן, ריצת הקוד צריכה להיות שונה בהתאם לביט שמוגרל.

- בדיקת איזומורפיות של גרפים (מטריצות שכנויות):
  בעזרת הפרמוטציה על הקודקודים, בדקנו אם כל כניסה במטריצה המקורית זהה לכניסה המתאימה במטריצה החדשה.

- בדיקה אם מסלול הוא מעגל המילטוני:
  בהינתן מסלול, בדקנו אם מתקיימים כל התנאים שנדרשים לקיום מעגל המילטוני.

מימוש הפרוטוקול

מימשנו את הפרוטוקול, והעלינו גם אותו לגיט [נספח 1: קישור לגיט של הפרויקט ב-GitHub].
המימוש מכיל את קבצי הפייתון עם השמות הבאים:
Prover, Verifier, CommonGraph, PrivateHamiltonianCycle, ElGamalCommitmentScheme.

הקובץ ElGamalCommitmentScheme.py [נספח 2: ElGamalCommitmentScheme.py] זהה לקובץ שהצגנו בפרוטוקול הקודם.

בקובץ PrivateHamiltonianCycle.py [נספח 7: PrivateHamiltonianCycle.py] שמורה רשימה עם המעגל ההמילטוני של הגרף שידוע למוכיח (הקלט הפרטי של המוכיח). בקובץ זה יש מתודה בשם **getPrivateHamiltonianCycle** שמחזירה את המעגל ההמילטוני הידוע.

הקובץ CommonGraph.py [נספח 8: CommonGraph.py] דומה מאוד לקובץ עם אותו השם שיצרנו בפרוטוקול הקודם. בקובץ הזה מיוצר הגרף המכוון שעליו מפעילים את הפרוטוקול (הקלט המשותף של המוכיח והמוודאת) כמטריצת שכנויות. בקובץ הזה יש מתודה בשם **getCommonGraph** שמחזירה את הגרף, והיא משמשת את המוכיח והמוודאת.
כמו כן, בקובץ יש מספר מתודות שמשמשות ליצירת גרפים שונים ולייצוגם כמטריצת שכנויות, כמו: createDirectedGraphByAdjMatrix (מתודה שמקבלת מהמשתמש רצף של קשתות ויוצרת מטריצת שכנויות שמייצגת את הגרף המכוון) ו-loadDirectedGraphAsAdjMatrix (מתודה שמקבלת קובץ טקסט שמכיל קשתות של גרף מכוון, ויוצרת ממנו מטריצת שכנויות שמייצגת את הגרף).

בקובץ Prover.py [נספח 9: Prover.py] מתבצע החלק של המוכיח בפרוטוקול. נסביר את המתודות המרכזיות שהוא מכיל:

- **start_server** – מתודה שמייצרת את התקשורת בין המוכיח למוודאת, וקוראת למתודה **handle_client** שבה מתבצע הצד של המוכיח בפרוטוקול.
- **firstStepOfProver** – מתודה שמקבלת את ה-socket, את הערכים הידועים ואת הייצוג של הגרף, ומבצעת את הצעד הראשון של המוכיח לפי הפרוטוקול (הפעלת פרמוטציה על הצמתים וקבלת גרף חדש בהתאם, וכן חישוב ההתחייבות על כל אחת מהכניסות של המטריצה שמייצגת את הגרף החדש).

- **secondStepOfProver** – מתודה שמקבלת את ה-socket, את הייצוג של הגרף החדש, את הפרמוטציה, את הערכים הרנדומליים שמשמשים לפתיחה, את הביט שנבחר ואת המעגל ההמילטוני. המתודה מבצעת את הצעד השני של המוכיח לפי הפרוטוקול (אם הביט הוא 0 המוכיח שולח למוודאת את הפרמוטציה ואת פתיחת ההתחייבויות של כל אחת מכניסות המטריצה החדשה. אם הביט הוא 1 המוכיח שולח למוודאת את החשיפה של הכניסות שמתאימות לקשתות של המעגל ההמילטוני בגרף החדש).

בקובץ Verifier.py **[נספח 10: Verifier.py]** מתבצע החלק של המוודאת בפרוטוקול. נסביר את המתודות המרכזיות שהוא מכיל:

- **start_client** – מתודה שמייצרת את החיבור של הלקוח לשרת עבור התקשורת בין המוכיח למוודאת, ולאחר מכן מבצעת את הצד של המוודאת בפרוטוקול.
- **firstStepOfVerifier** – מתודה שמקבלת את ה-socket, ומבצעת את הצעד הראשון של המוודאת לפי הפרוטוקול (הגרלת ביט באקראי ושליחתו למוכיח).
- **secondStepOfVerifier** – מתודה שמקבלת את ה-socket, את הביט הנבחר, את הערכים הידועים, את הפתיחה שהתקבלה מהמוכיח (בהתאם לביט שנבחר), את מערך ההתחייבויות של כל כניסות המטריצה ואת הייצוג של הגרף. המתודה מבצעת את הצעד השני של המוודאת לפי הפרוטוקול (בדיקה שפתיחת התחייבויות היא תקינה, וכן בדיקה אם הגרף החדש שנחשף והגרף המקורי הם איזומורפיים [אם הביט הוא 0] או בדיקה שהמסלול שנחשף הוא מעגל המיליטוני [אם הביט הוא 1], והחלטה על קבלת הטענה או דחייתה בהתאם לפרוטוקול).

בדומה לפרוטוקול הקודם, גם לצורך המימוש הזה נעזרנו במספר ספריות עיקריות: הספריות socket ו-threading שימשו למימוש התקשורת בין המוכיח ומוודאת, הספרייה random שימשה לבחירת ביט באקראי ולביצוע פעולות שונות בסכמת ההתחייבות והספרייה numpy שימשה ליצירת פרמוטציה אקראית על הצמתים.
לצורך הרצת הפרוטוקול, מריצים את הקובץ Prover.py ולאחריו את הקובץ Verifier.py.
בתחילה, נוצרת התקשורת בין המוכיח והמוודאת, ולאחר מכן הריצה ממשיכה לפי שלבי הפרוטוקול.

ניסויים עבור ה-ZKP למעגל המילטוני בגרף

ניסויים לבדיקת נכונות המימוש

בתחילה, רצינו לבדוק את נכונות המימוש. כלומר, רצינו לראות שהתרגום של הפרוטוקול לתוכנית שיצרנו הוא נכון ולא מכיל שגיאות.

לצורך כך, יצרנו מספר גרפים שחלקם בעלי מעגל המילטוני וחלקם לא, הגדרנו להם מסלולים שחלקם אכן מעגלים המילטוניים וחלקם לא, ועבור כל אחד מהגרפים ומהמסלולים שמתאימים להם הרצנו את התוכנית מספר פעמים וראינו שהתוצאות אכן נכונות.

נראה דוגמה לאחד מהניסויים האלה:

הגרף המשותף

```
# A directed graph with hamiltonian cycle which has 20 vertices and
60 edges
M2 = [[0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 1,
0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 1, 0,
1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0], [1, 0, 0, 1, 0, 0,
0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1, 0, 1, 0, 0,
0, 0, 0, 0, 0, 0, 1, 0, 0, 0], [0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0,
0, 0, 0, 1, 0, 0], [0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0,
1, 0], [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1], [0, 0,
0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0], [0, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1], [0, 0, 0, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 0, 1, 0, 1, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 1, 0, 1], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 1, 0, 0, 1, 0]]
```

המעגל ההמילטוני בגרף (הקלט הפרטי של המוכיח)

```
hamiltonianCycle2 = [0, 1, 2, 9, 8, 7, 6, 5, 14, 13, 12, 19, 15, 16,
17, 18, 10, 11, 3, 4, 0]  # an hamiltonian cycle in M2
```

M2 מכיל את ייצוג של גרף מכוון כמטריצת שכנויות ו-*hamiltonianCycle2* מכיל את הייצוג של מעגל המילטוני בגרף הזה.

הפלט של ההרצה

```
Prover:

Perform the first step of the prover
The verifier chose the bit:  0
Perform the second step of the prover:
Reveal the permutation and the commitments of the new graph's entries

The verifier accepted the claim!
```

```
Verifier:

Perform the first step of the verifier
The chosen bit:  0
Perform the second step of the verifier:
Check if the decommits are valid and if the new graph is isomorphic to the original graph

Accept the claim - the graph has a hamiltonian cycle!

The interaction with the prover has finished
```

ניתן לראות שכל אחד מהמשתתפים בפרוטוקול מבצע את השלבים שלו. הביט שנבחר בהרצה זו הוא 0, ובהתאם לכך מתבצעות הפעולות המתאימות בפרוטוקול. בסופו של דבר, המוודאת קיבלה את הטענה והחליטה שבגרף יש מעגל המילטוני.

ניסויים לבדיקת יעילות המימוש ביחס לפרמטרים שונים

בדומה לפרוטוקול הקודם, רצינו לבדוק את אותם המדדים עבור התוכנית (זמן ריצה, צריכת זיכרון ותקשורת). הסיבה לכך היא שמדדים אלה הם המדדים העיקריים שמושפעים מהקלט ומגדירים את היעילות והביצועים של הקוד, וכן הבדיקה של אותם המדדים תעזור לנו להשוות בין הביצועים של שני הפרוטוקולים.
נרצה לבחון גם כן את ההשפעה של מספר הצמתים ואת ההשפעה של מספר הקשתות על כל אחד מהמדדים, להבין מה משפיע בצורה משמעותית על ערכי המדדים השונים בפרוטוקול ולהסיק עבור איזה מדדים ועבור איזה קלטים הפרוטוקול יעיל.

לצורך כך, השתמשנו באותם גרפים ששימשו לבדיקת המדדים בפרוטוקול הקודם.
נדגיש כי הגרפים האלה משמשים רק עבור בדיקת מדדי היעילות ולא ידוע לנו אם הם בעלי מעגל המילטוני.

לצורך בדיקת ההשפעה של מספר הקשתות, השתמשנו במתודה שיצרנו בפרוטוקול הקודם (*createCheckingGraphByEdges*). באופן דומה, לצורך בדיקת ההשפעה של מספר הצמתים השתמשנו גם כן במתודה שיצרנו בפרוטוקול הקודם (*createCheckingGraphByNodes*).
כמו כן, יצרנו מתודה בשם *loadCheckingDirectedGraphAsAdjMatrix* שמקבלת את הגישה לקובץ של תת גרף מכוון שנוצר, ושומרת אותו בייצוג של מטריצת שכנויות.

עבור בדיקת ההשפעה של מספר הקשתות, יצרנו מספר תתי-גרפים של גרף ה-*Facebook* עם מספר צמתים זהה ועם מספר קשתות שגדל בקפיצות קבועות.
בנוסף, לבדיקת ההשפעה של מספר הצמתים, יצרנו מספר תתי-גרפים של גרף ה-*Gnutella* עם מספר צמתים שגדל (ובהתאם לכך גם מספר הקשתות גדל) בקפיצות קבועות.

עבור הניסויים עבדנו בצורה זהה לניסויים שבוצעו עבור הפרוטוקול ל-3 צביעות של גרף (השתמשנו באותן ספריות, הוספנו פקודות לקבלת ערכי המדדים, ביצענו מספר מדידות ויצרנו גרפים בהתאם לתוצאות). הגרפים ישמשו להבנת הסיבוכיות האסימפטוטית של ערכי המדדים כתלות במספר הצמתים ובמספר הקשתות.

בתחילה, יצרנו תתי-גרפים וביצענו ניסויים עבור כל ביט בנפרד (0 או 1), כדי שנוכל להבין גם את ההשפעה של בחירת הביט על ערכי המדדים.
לאחר מכן, יצרנו תתי-גרפים וביצענו מספר הרצות לכל מדד ולכל תת-גרף עם האקראיות של הביט (כמו שהפרוטוקול אמור לרוץ), חישבנו את ערכי המדדים המשוקללים הממוצעים, וקיבלנו עבורם תוצאות.

<u>תוצאות הניסויים – השפעת מספר הצמתים (בהתאם לערך הביט):</u>

לאחר ביצוע המדידות לבדיקת ההשפעה של מספר הצמתים, ריכזנו את ערכי המדדים שהתקבלו בטבלאות הבאות:

$bit = 0$:

מספר הצמתים (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
150	10.848	11.101	3.599	8.949	12.793
300	42.985	43.15	14.373	37.042	50.098
450	97.823	101.67	32.353	81.562	111.637
600	176.252	180.865	57.536	146.113	199.77
750	271.017	276.378	89.284	224.391	310.551
900	398.872	400.906	129.32	323.551	331.391
1050	540.548	564.724	175.802	597.339	335.742
1200	705.726	728.6	227.455	906.113	429.911
1350	909.658	976.778	291.633	1162.028	559.773
1500	1094.411	1176.537	359.893	1436.485	675.656

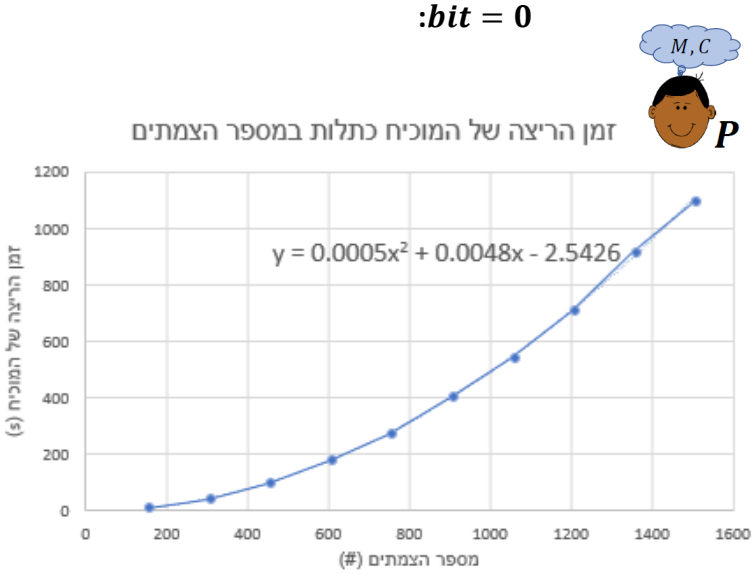
$bit = 1$:

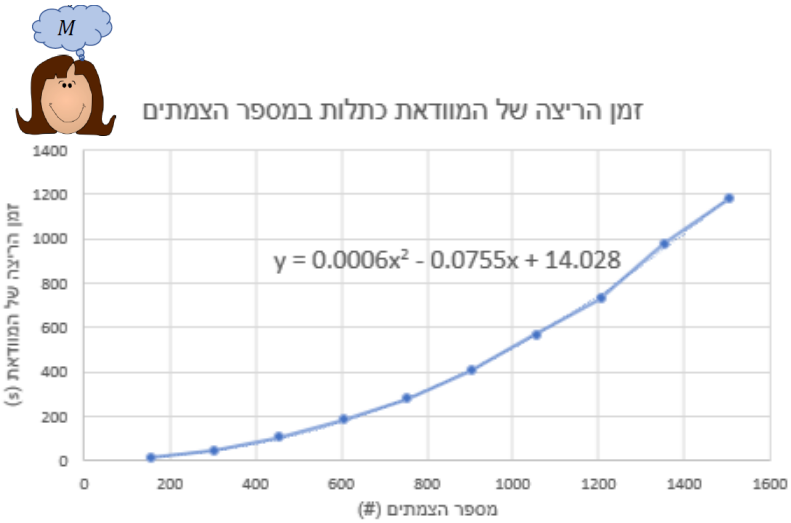
מספר הצמתים (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
150	10.37	0.175	1.824	6.461	6.238
300	42.919	0.699	7.168	21.508	21.977
450	99.214	1.585	16.286	48.469	49.445
600	167.991	2.946	28.685	85.078	87.168
750	275.809	4.642	45.142	131.949	134.883
900	380.2	6.897	64.811	190.52	152.102
1050	528.54	11.586	88.423	450.742	164.992
1200	667.721	25.144	114.332	636.519	217.809
1350	896.741	34.032	146.379	817.7	278.215
1500	1064.394	56.441	180.363	1065.743	345.703

<u>זמן ריצה</u>

מהמדידות של זמני הריצה קיבלנו את הגרפים הבאים:

$bit = 0$:





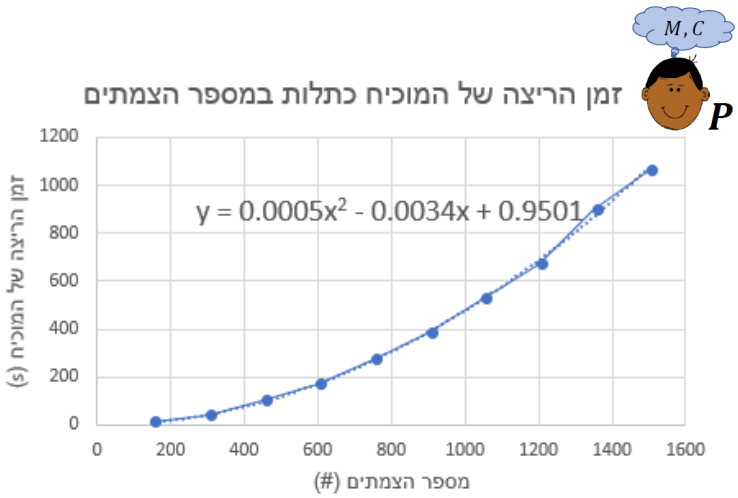
ניתן לראות שזמני הריצה של המוכיח והמוודאת עולים ככל שמספר הצמתים עולה.
נשים לב שזמני הריצה הם איטיים ביחס לתוצאות שנראה עבור $bit = 1$, ושעבור גרפים עם מאות צמתים המוכיח והמוודאת מסיימים את ריצתם רק לאחר מספר דקות.

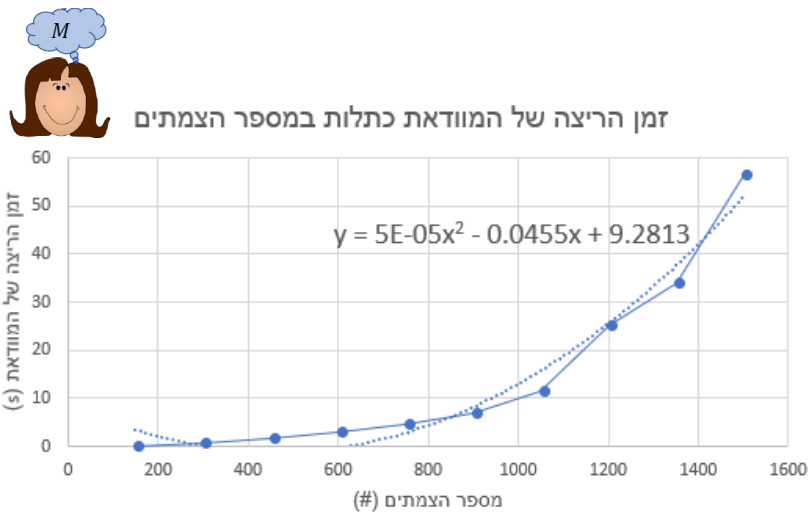
כמו כן, זמן הריצה של המוודאת הוא קצת יותר גדול מזה של המוכיח, וזאת מכיוון שהמוודאת מבצעת מספר גדול יותר של פעולות "כבדות", כמו: בדיקת איזומורפיות של הגרף החדש והגרף המקורי.

הגרף של המוכיח הוא בקירוב פולינומי מסדר 2. הקריאה והייצוג של הגרף במטריצת שכנויות לוקחים זמן של $O(|E| + |U|^2)$. בנוסף, המוכיח מבצע מספר פעולות שמתבצעות בסיבוכיות זמן של $O(|U^2|)$ (כמו: יצירת מטריצה חדשה, חישוב ההתחייבות על כל אחת מהכניסות של המטריצה והחשיפה שלהן).

הגרף של המוודאת הוא גם כן בקירוב פולינומי מסדר 2. הסיבה לכך היא שגם כאן יש השפעה זהה לקריאה והייצוג של הגרף ושהמוודאת גם מבצעת מספר פעולות נוספות שמתבצעות בסיבוכיות זמן של $O(|U^2|)$ (כמו: קבלת ההתחייבויות, יצירת המטריצה החדשה ובדיקת האיזומורפיות של הגרפים).

$\boldsymbol{bit = 1}$:





ניתן לראות שהגרפים של המוכיח והמוודאת הם עדיין בקירוב פולינומיים מסדר 2 בגלל שבכל אחד מהם עדיין יש פעולות שמתבצעות בזמן ריבועי (כמו: הקריאה והייצוג של הגרף, יצירת מטריצה חדשה, חישוב ההתחייבויות על כל הכניסות של המטריצה וקבלת ההתחייבויות).

עם זאת, נשים לב שזמני הריצה הם יותר נמוכים.

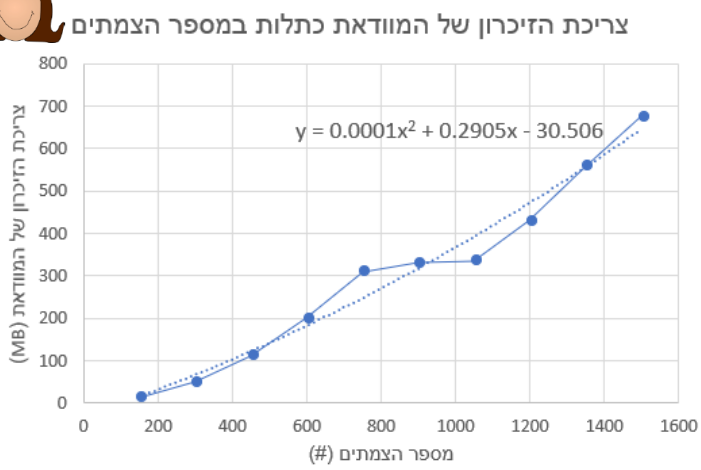
זמן הריצה של המוודאת הוא הרבה יותר מהיר מכיוון שהרבה פעולות שמתבצעות ע"י המוודאת תלויות בביט שנבחר (הקבלה והאימות של החשיפות והבדיקה הרלוונטית תלויות בביט שנבחר), ועבור $bit = 1$ הפעולות מהירות יותר (כמו: בדיקה אם מסלול הוא המילטוני לעומת בדיקת איזומורפיות של גרפים).

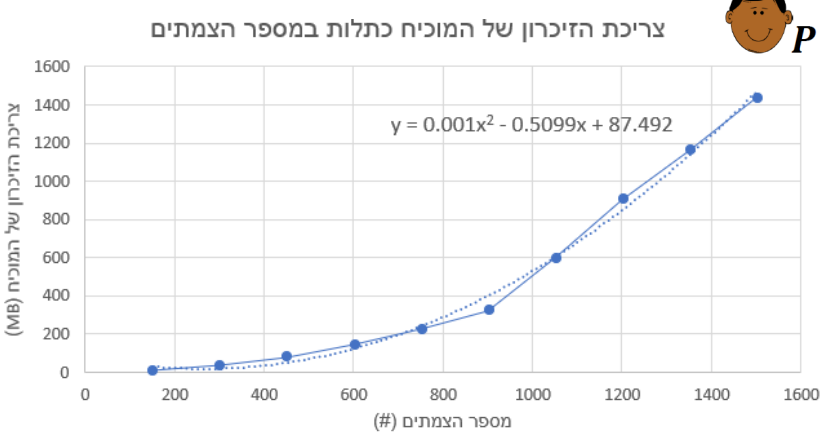
עבור המוכיח חלק מהפעולות הן יותר מהירות (הביט שנבחר משפיע על מה שהמוכיח יחשוף), אבל בצורה פחות משמעותית. לכן, זמן הריצה שלו הוא רק מעט יותר קטן ביחס למקרה שבו $bit = 0$.

<u>צריכת הזיכרון</u>

מהמדידות של צריכת הזיכרון קיבלנו את הגרפים הבאים:

$\boldsymbol{bit = 0}$:



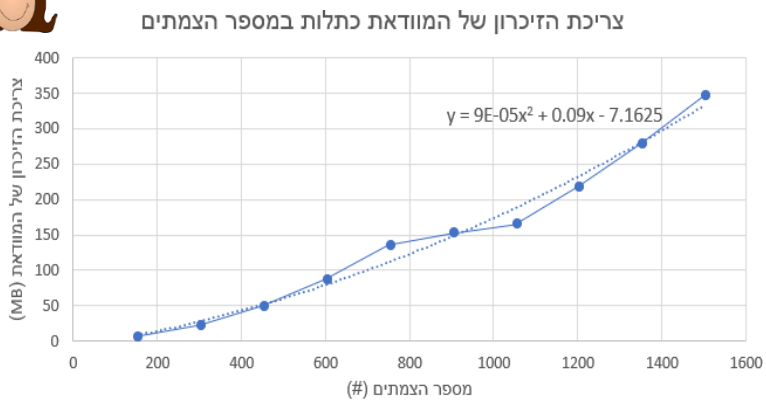


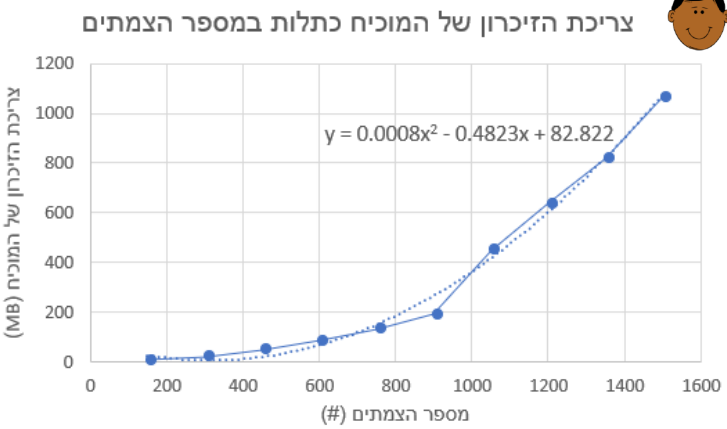
ניתן לראות שצריכת הזיכרון של המוכיח והמוודאת עולות ככל שמספר הצמתים עולה.

הגרף של המוכיח הוא בקירוב פולינומי מסדר 2, מכיוון שהוא שומר הרבה אובייקטים שדורשים זיכרון של $O(|U^2|)$ (כמו: הייצוגים של הגרף המקורי והחדש, ההתחייבויות על הכניסות של המטריצה החדשה והחשיפה שלהן).

הגרף של המוודאת הוא גם כן בקירוב פולינומי מסדר 2, מכיוון שהיא גם שומרת אובייקטים שדורשים זיכרון של $O(|U^2|)$ (כמו: הייצוגים של הגרפים, ההתחייבויות והחשיפה שלהן).

: $\boldsymbol{bit = 1}$





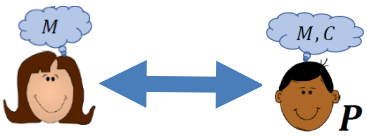
הגרפים של המוכיח והמוודאת הם עדיין בקירוב פולינומיים מסדר 2, מכיוון שעדיין יש להם אובייקטים שדורשים זיכרון של $O(|U^2|)$ (כמו: הייצוגים של הגרפים וההתחייבויות).

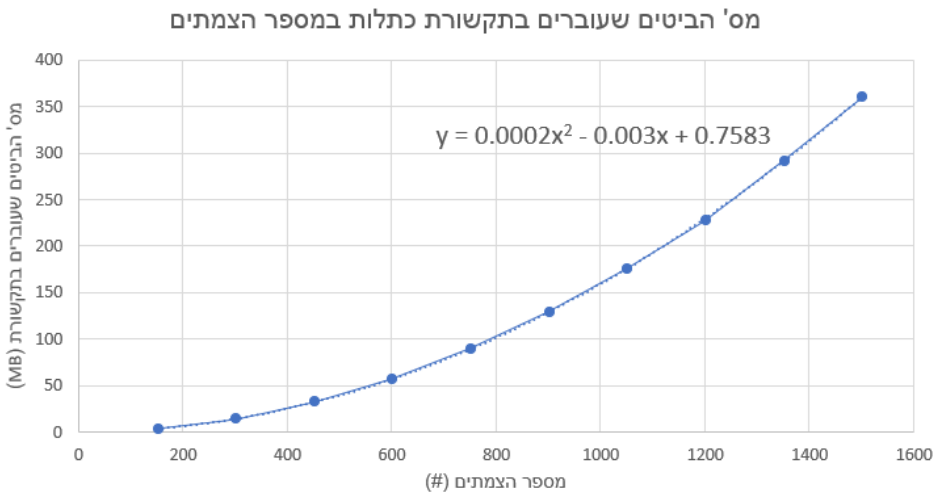
עם זאת, צריכת הזיכרון של המוכיח והמוודאת הם יותר קטנים מכיוון שעבור $bit = 1$ צריך לשמור אובייקטים שדורשים פחות זיכרון (כמו: המסלול ההמילטוני של הגרף החדש לעומת כל המטריצה שמייצגת את הגרף החדש).

<u>תקשורת</u>

מהמדידות של מס' הביטים שעוברים בתקשורת קיבלנו את הגרף הבא:

: $\boldsymbol{bit = 0}$



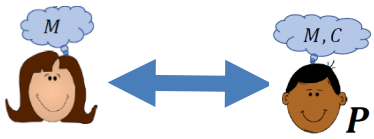


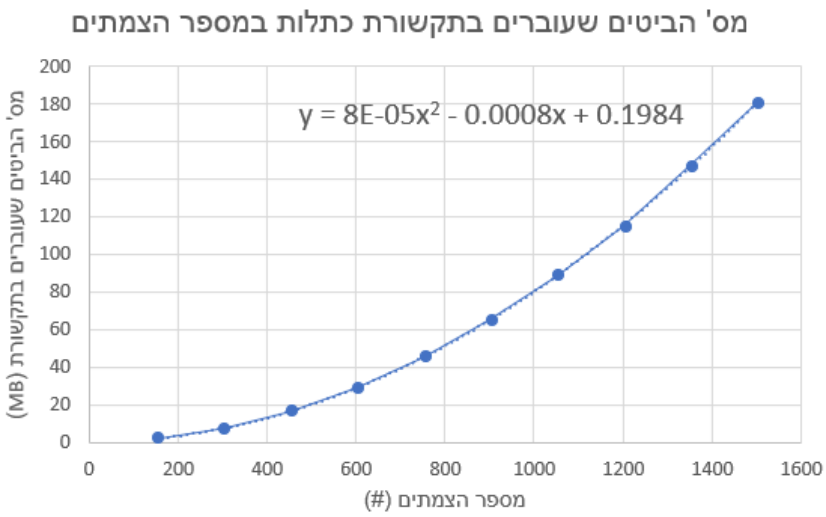
ניתן לראות שמספר הביטים שעוברים בתקשורת בין המוכיח למוודאת עולה ככל שמספר הצמתים עולה.

נשים לב כי מספר הביטים שעוברים בתקשורת בין המוכיח למוודאת הוא פולינומי מסדר 2 ביחס למספר הצמתים - $O(|U|^2)$. זאת מכיוון שבתקשורת מועברות ההתחייבויות על כל אחת מהכניסות של מטריצת השכנויות.
כמו כן, בשלב החשיפה, עבור $bit = 0$ מועברות הפרמוטציה ופתיחת כל ההתחייבויות על כניסות המטריצה (זה גם כן דורש זיכרון של $O(|U|^2)$).

שאר הנתונים שמועברים בתקשורת (הביט שנבחר והתוצאה [ההחלטה של המוודאת]) דורשים זיכרון קבוע - $O(1)$.

: $\boldsymbol{bit = 1}$





הגרף שהתקבל הוא עדיין פולינומי מסדר 2 ביחס למספר הצמתים - $O(|U|^2)$, מכיוון שבתקשורת מועברות ההתחייבויות על כל אחת מהכניסות של מטריצת השכנויות.

בשלב החשיפה, עבור $bit = 1$ מועבר המעגל ההמילטוני עם החשיפות של הכניסות שמתאימות לו (זה דורש זיכרון של $O(|U|)$).

בנוסף, בדומה למקרה הקודם, שאר הנתונים שמועברים בתקשורת (הביט שנבחר והתוצאה [ההחלטה של המוודאת]) דורשים זיכרון קבוע - $O(1)$.

<u>תוצאות הניסויים – השפעת מספר הקשתות (בהתאם לערך הביט):</u>

לאחר ביצוע המדידות לבדיקת ההשפעה של מספר הקשתות, ריכזנו את ערכי המדדים שהתקבלו בטבלאות הבאות:

$bit = 0$:

מספר הקשתות (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
1000	470.341	483.637	159.588	431.36	317.785
2000	470.75	486.208	159.971	449.316	304.743
3000	486.668	502.978	160.159	415.215	324.422
4000	483.166	497.166	160.109	404.421	304.957
5000	502.904	508.824	159.857	410.491	304.895
6000	496.818	521.352	158.555	403.664	304.903
7000	488.611	503.182	160.395	400.735	305.071
8000	477.977	489.718	160.328	441.27	304.95
9000	482.701	497.625	160.572	438.156	328.863

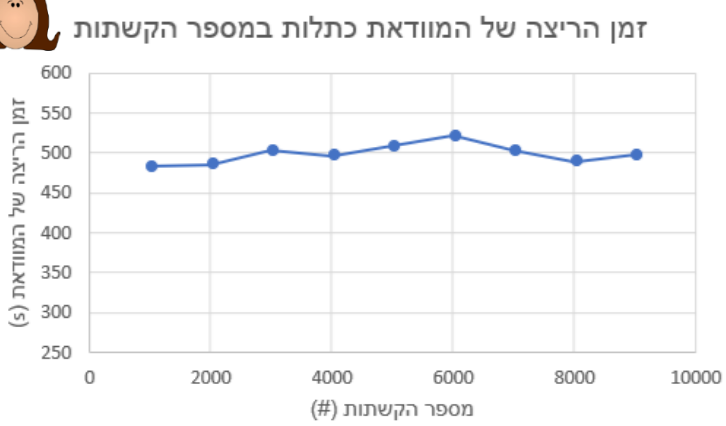
$bit = 1$:

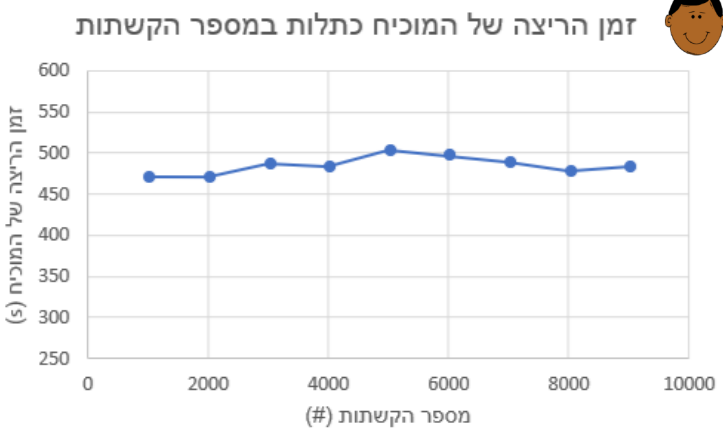
מספר הקשתות (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
1000	498.401	10.455	80.091	235.144	168.379
2000	493.245	12.754	79.957	254.555	149.117
3000	488.117	11.314	79.465	236.196	149.465
4000	500.093	12.189	80.296	246.797	149.855
5000	501.103	12.258	80.389	273.801	149.609
6000	496.318	12.141	80.479	275.656	149.344
7000	489.502	10.281	79.719	242.34	179.707
8000	490.693	11.709	80.103	260.75	149.727
9000	489.318	10.918	80.583	235.703	182.598

<u>זמן ריצה</u>

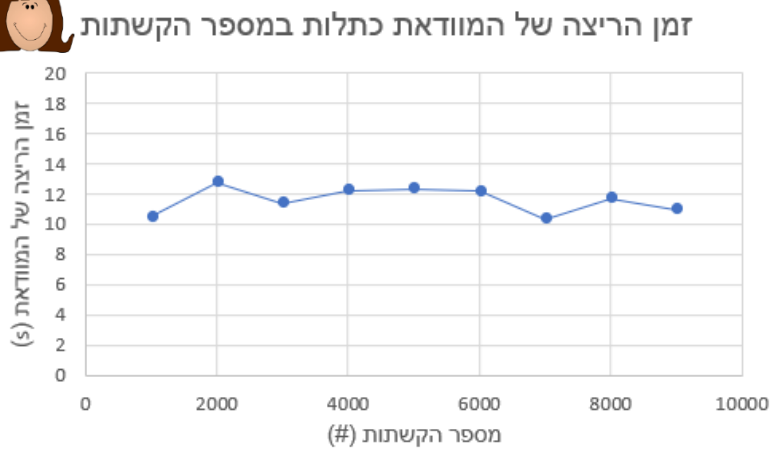
מהמדידות של זמני הריצה קיבלנו את הגרפים הבאים:

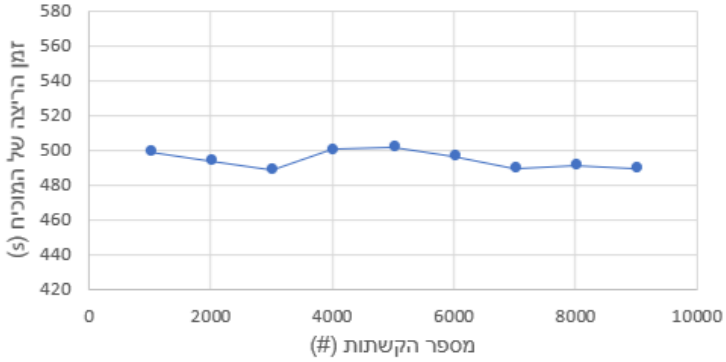
$bit = 0$:





$: bit = 1$





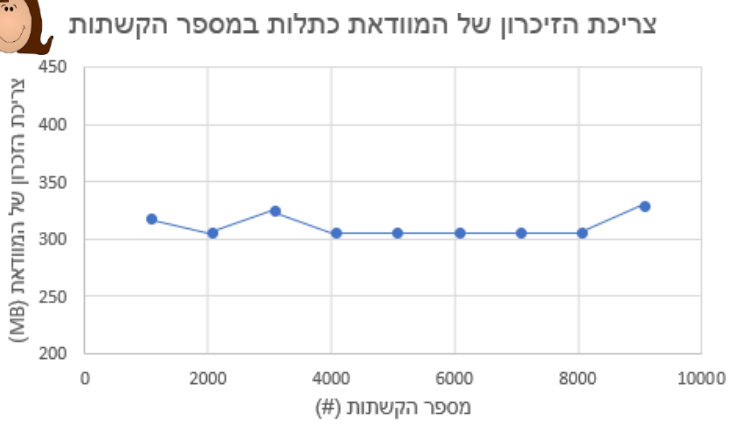
ניתן לראות שזמני הריצה של המוכיח והמוודאת קבועים בקירוב ככל שמספר הקשתות עולה (מספר הצמתים קבוע).

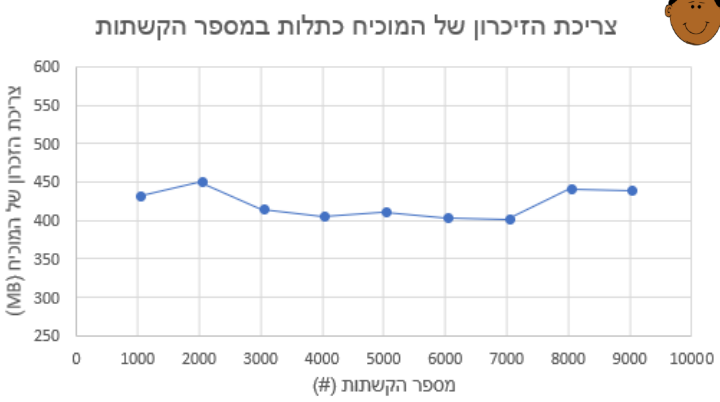
זמני הריצה של הפעולות שמתבצעות בריצת המוכיח והמוודאת (כמו: קריאת הגרף, הפעלת הפרמוטציה על הקודקודים ויצירת גרף חדש, חישוב ההתחייבויות על הכניסות של המטריצה והחשיפה שלהן) תלויים בעיקר במספר הצמתים ולא במספר הקשתות.
כלומר, זמני הריצה של המוכיח והמוודאת אינם מושפעים בצורה משמעותית מכמות הקשתות, ולכן הגרפים של המוכיח ושל המוודאת הם קבועים בקירוב.

<u>צריכת הזיכרון</u>

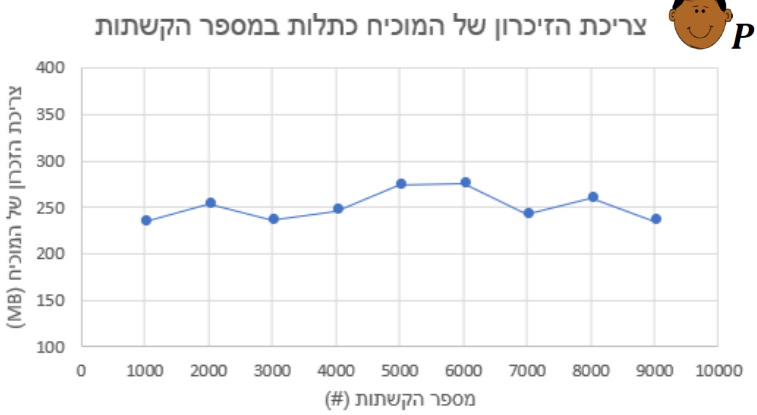
מהמדידות של צריכת הזיכרון קיבלנו את הגרפים הבאים:

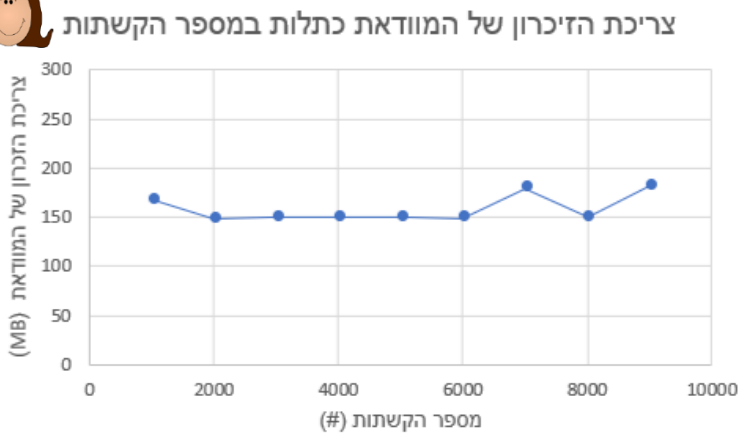
$: bit = 0$





$: bit = 1$





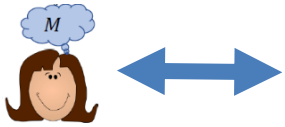
ניתן לראות שצריכת הזיכרון של המוכיח ושל המוודאת קבועות בקירוב ככל שמספר הקשתות עולה.

צריכת הזיכרון במהלך הריצה של המוכיח ושל המוודאת (כמו: הזיכרון עבור שמירת הייצוג של הגרפים, המעגל ההמילטוני, הפרמוטציה על הקודקודים, ההתחייבויות על הכניסות של המטריצה והחשיפה שלהן) תלוי בעיקר במספר הצמתים ולא במספר הקשתות.
כלומר, צריכת הזיכרון של המוכיח והמוודאת אינן מושפעות בצורה משמעותית מכמות הקשתות, ולכן הגרפים של המוכיח ושל המוודאת הם קבועים בקירוב.

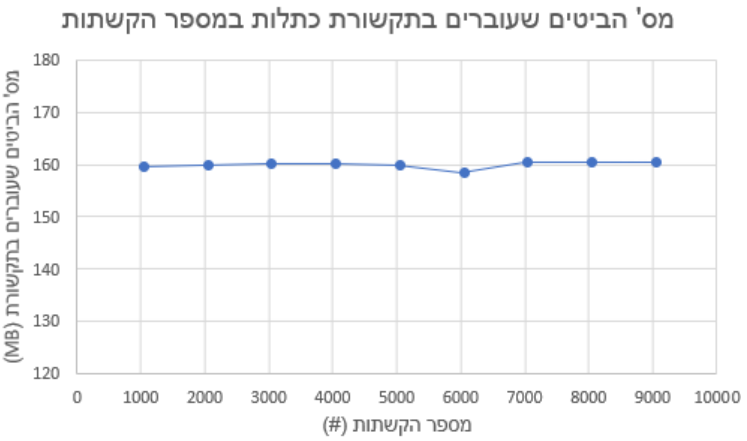
<u>תקשורת</u>

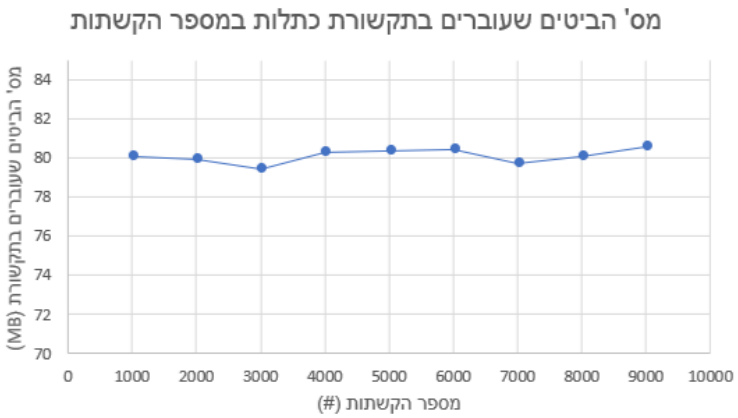
מהמדידות של מס' הביטים שעוברים בתקשורת קיבלנו את הגרפים הבאים:

$: bit = 0$



$: bit = 1$





ניתן לראות שמספר הביטים שעוברים בתקשורת בין המוכיח למוודאת הוא קבוע בקירוב ככל שמספר הקשתות עולה.

כפי שראינו קודם, מספר הביטים שעוברים בתקשורת במהלך הריצה של המוכיח והמוודאת תלוי רק במספר הצמתים ולא במספר הקשתות (בתקשורת מועברים ההתחייבויות על כל אחת מהכניסות של מטריצת השכנויות, הביט שנבחר, החשיפה של ההתחייבויות [בהתאם לערך הביט] והתוצאה [ההחלטה של המוודאת], וכל אלה אינם תלויים במספר הקשתות). כלומר, מספר הביטים שעוברים בתקשורת בין המוכיח למוודאת אינו מושפע מכמות הקשתות, ולכן הגרף הוא קבוע בקירוב.

תוצאות הניסויים – המדדים המשוקללים הממוצעים

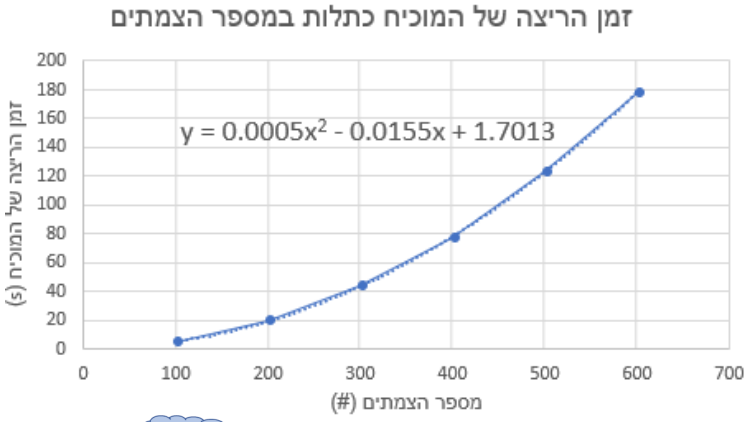
ביצענו מספר הרצות לכל מדד ולכל תת-גרף (עבור תתי-הגרפים שמשמשים לבדיקת ההשפעה של מספר הצמתים ומספר הקשתות) עבור בחירה אקראית של הביט b, כאשר ידוע שהביט שמוגרל מקבל את הערך 0 או 1 בהסתברות $\frac{1}{2}$ לכל אחד מהם (עבור טבלאות הערכים שהתקבלו מכל הרצה ראו **[נספח 11: טבלאות עם ערכי המדדים שהתקבלו מההרצות עבור חישוב המדדים המשוקללים הממוצעים]**). חישבנו את ערכי המדדים המשוקללים הממוצעים, וקיבלנו עבורם תוצאות.

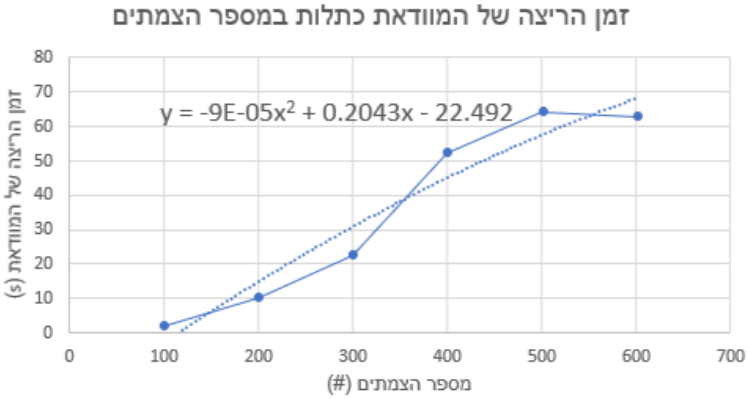
השפעת מספר הצמתים:

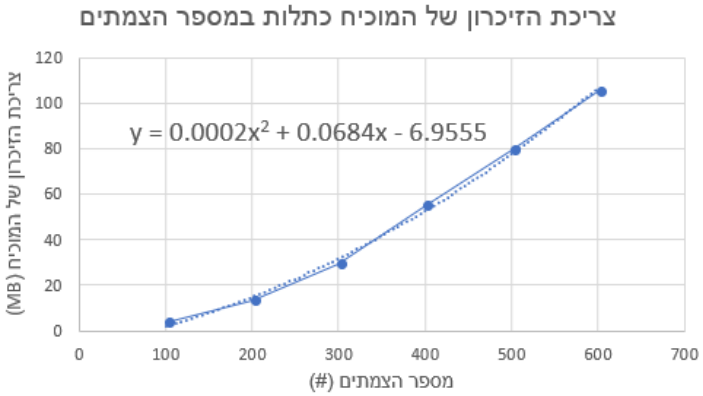
לאחר ביצוע המדידות לבדיקת ההשפעה של מספר הצמתים, חישבנו את ערכי המדדים המשוקללים הממוצעים, וריכזנו אותם בטבלה הבאה:

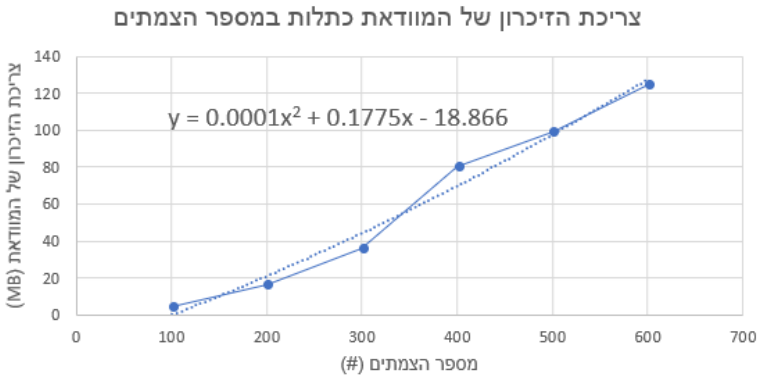
מספר הצמתים (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
100	5.0845	1.664666667	1.076166667	3.372833333	4.264333333
200	19.49966667	10.18416667	4.814166667	13.2315	16.459
300	43.526	22.305	10.779	29.09983333	36.008
400	77.16883333	52.0975	21.34666667	55.22533333	80.39116667
500	122.6266667	64.10833333	30.03266667	79.3795	99.425
600	177.4486667	62.73716667	38.38183333	105.2095	124.5305

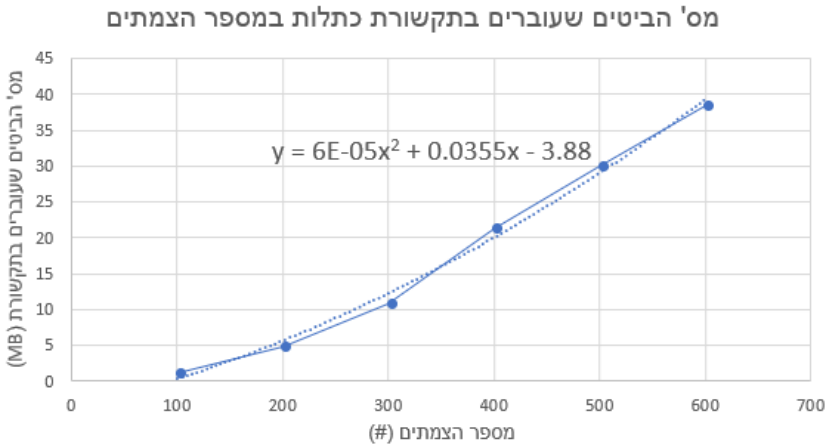
מהערכים שהתקבלו, קיבלנו את הגרפים הבאים:

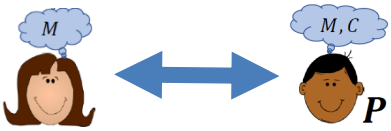












ניתן לראות שעדיין מספר הצמתים משפיע בצורה ניכרת, בכך שככל שמספר הצמתים עולה, כל אחד מערכי המדדים עולה.

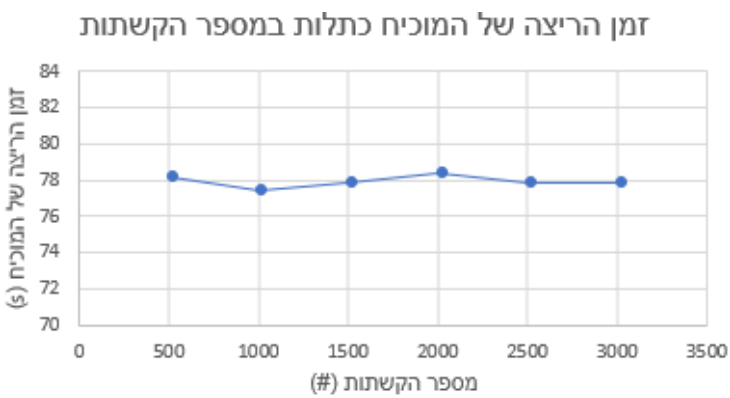
כמו כן, נשים לב שהגרפים עבור זמני הריצה של המוודאת ועבור צריכת הזיכרון שלה נראים קצת פחות רציפים מהשאר. הסיבה לכך היא שהביט שנבחר משפיע בצורה משמעותית על זמן הריצה של המוודאת ועל צריכת הזיכרון שלה.
אם ערך הביט הוא 1, זמן הריצה של המוודאת וצריכת הזיכרון שלה הם נמוכים יותר בצורה משמעותית מהערכים המקבילים עבור המקרה שבו ערך הביט הוא 0.
ככל שיתבצעו יותר ריצות ובהתאם אליהם יחושבו הערכים המשוקללים הממוצעים, הגרפים יתכנסו יותר לגרפים רציפים עם מגמה פולינומית מסדר 2 (ידוע כי בכל הרצה ערך הביט מוגרל באקראי לערך 0 או 1).

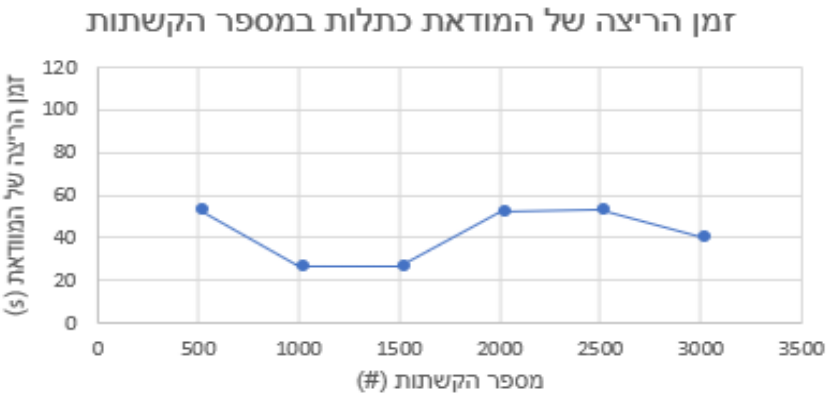
<u>השפעת מספר הקשתות:</u>

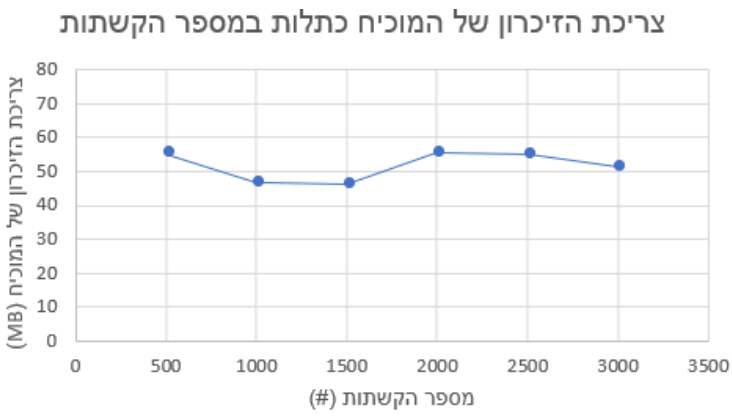
לאחר ביצוע המדידות לבדיקת ההשפעה של מספר הקשתות, חישבנו את ערכי המדדים המשוקללים הממוצעים, וריכזנו אותם בטבלה הבאה:

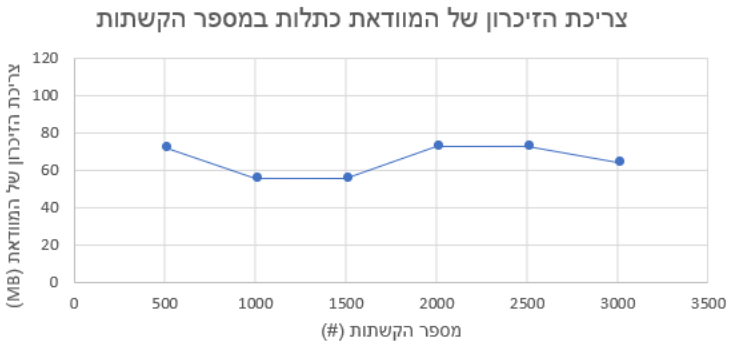
מספר הקשתות (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
500	78.14216667	53.431	21.3625	55.32966667	72.22083333
1000	77.40633333	26.70333333	17.14783333	46.765	55.81633333
1500	77.81133333	26.66783333	17.08266667	46.37333333	55.53766667
2000	78.378	52.57933333	21.451	55.44583333	72.49666667
2500	77.82233333	53.25883333	21.46633333	55.284	72.457
3000	77.831	40.6365	19.41133333	51.27916667	64.17183333

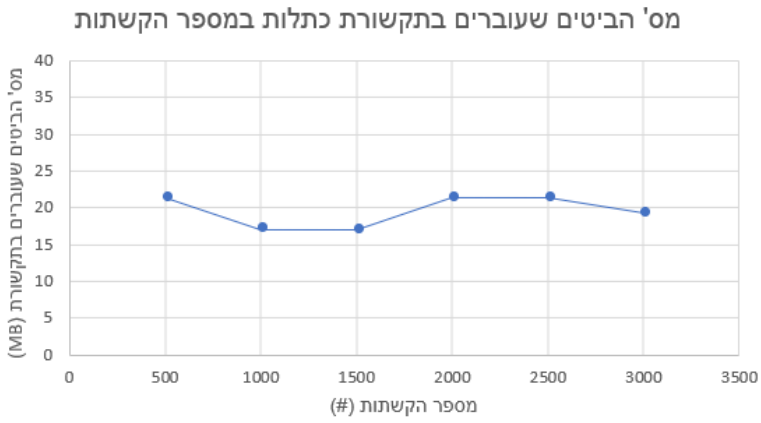
מהערכים שהתקבלו, קיבלנו את הגרפים הבאים:

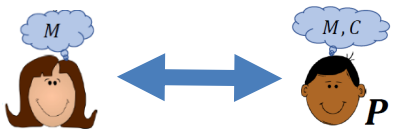












ניתן לראות שעדיין למספר הקשתות אין השפעה משמעותית על ערכי המדדים, כל אחד מהגרפים הוא בקירוב קבוע.

בדומה למה שראינו עבור השפעת מספר הצמתים, נשים לב שגם כאן הגרפים עבור זמני הריצה של המוודאת ועבור צריכת הזיכרון שלה נראים קצת פחות רציפים משאר הגרפים מאותה סיבה שציינו.
ככל שיתבצעו יותר ריצות ובהתאם אליהם יחושבו הערכים המשוקללים הממוצעים, הגרפים יתכנסו יותר לגרפים קבועים (בכל הרצה ערך הביט מוגרל באקראי לערך 0 או 1).

תיארנו את המימוש, את הניסויים ואת התוצאות שהתקבלו עבור ה-ZKP למעגל המילטוני בגרף. מהתוצאות שהתקבלו, הסקנו מסקנות בנוגע לנכונות הקוד ולהשפעה של מספר הצמתים, מספר הקשתות וערך הביט על הערכים של המדדים השונים.

השוואה בין הפרוטוקולים ומסקנות

לאחר ההבנה והמימוש של הפרוטוקולים, ביצוע הניסויים וניתוח התוצאות, ניתן להשוות בין הפרוטוקולים ולהסיק מסקנות שונות:

- <u>השוואת ערכי המדדים בניסויים השונים</u>:

  מתוצאות הניסויים ניתן להסיק שערכי כל המדדים שמדדנו (זמני הריצה, צריכת הזיכרון ומספר הביטים שעוברים בתקשורת) הם הרבה יותר נמוכים עבור ה-ZKP ל-3 צביעות של גרף בהשוואה לערכים של אותם מדדים עבור ה-ZKP למעגל המילטוני בגרף.

  לדוגמה, נוכל להסתכל על ערכי המדדים שהתקבלו עבור הרצת הפרוטוקולים עם מספר דומה של צמתים:

	מספר הצמתים (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
3-Col	1000	0.598	0.109	0.153	10.102	10.637
HC (bit=0)	1050	540.548	564.724	175.802	597.339	335.742

ניתן לראות שעבור מספר דומה מאוד של צמתים, התוצאות שהתקבלו עבור ה-ZKP למעגל המילטוני בגרף הן גדולות בכמה סדרי גודל מאלו שהתקבלו עבור ה-ZKP ל-3 צביעות של גרף.

ההבדל בזמני הריצה נובע מכך שב-ZKP למעגל המילטוני בגרף המוכיח והמוודאת מבצעים הרבה יותר פעולות שמתבצעות בסיבוכיות זמן של $O(|U^2|)$ (כמו: יצירת מטריצה חדשה, חישוב ההתחייבות על כל אחת מהכניסות של המטריצה, קבלת ההתחייבויות, בדיקת איזומורפיות של גרפים ועוד) ביחס ל-ZKP ל-3 צביעות של גרף.

ההבדל בצריכת הזיכרון נובע מכך שב-ZKP למעגל המילטוני בגרף המוכיח והמוודאת שומרים הרבה יותר אובייקטים שדורשים סיבוכיות זיכרון של $O(|U^2|)$ (כמו: הייצוגים של הגרף המקורי והחדש, ההתחייבויות על הכניסות של המטריצה החדשה והחשיפה שלהן) ביחס ל-ZKP ל-3 צביעות של גרף.

ההבדל במספר הביטים שעוברים בתקשורת נובע מכך שעבור ה-ZKP למעגל המילטוני בגרף הסיבוכיות של מספר הביטים שעוברים בתקשורת היא $O(|U|^2)$ (בעיקר כי מועברות ההתחייבויות על כל הכניסות של המטריצה) ואילו עבור ה-ZKP

ל-3 צביעות של גרף הסיבוכיות של מספר הביטים שעוברים בתקשורת היא $O(|U|)$ (מועברות ההתחייבויות על הצבעים של כל אחד מהצמתים).

- <u>דטרמיניסטיות הריצה והאקראיות במהלכה</u>:

ה-ZKP ל-3 צביעות של גרף משלב אקראיות בתהליך שבו המוודאת בוחרת קשת. הקשת נבחרת באקראי, ולאחר מכן ריצת הפרוטוקול ממשיכה באופן דטרמיניסטי.

לעומת זאת, ב-ZKP למעגל המילטוני בגרף האקראיות מתרחשת בתהליך שבו המוודאת מגרילה ביט, וערך הביט שהוגרל משפיע על סוג הפעולות שיתבצעו בהמשך הפרוטוקול. כלומר, כאשר נריץ את הפרוטוקול פעם אחר פעם, נקבל בכל ריצה פעולות שונות שיתבצעו ע״י המוכיח והמוודאת (החשיפה שהמוכיח שולח למוודאת והבדיקות שהמוודאת מבצעת משתנות בהתאם לערך הביט), ולכן ריצת הפרוטוקול אינה דטרמיניסטית.
בנוסף, ערכי המדדים השונים שציינו גם משתנים בצורה משמעותית בהתאם לביט שנבחר. ראינו שכאשר ערך הביט הוא 1, הפעולות שמתבצעות הן פחות מורכבות ובשל כך ערכי המדדים הם קטנים יותר.

- <u>השפעת הקלט על ערכי המדדים</u>:

מתוצאות הניסויים שהתקבלו עבור שני הפרוטוקולים, ניתן לראות שיש השפעה רבה למספר הצמתים על המדדים השונים (זמני הריצה, צריכת הזיכרון ומספר הביטים שמועברים בתקשורת). ככל שמספר הצמתים עולה, כך ערכי המדדים עולים.
לעומת זאת, למספר הקשתות אין כמעט השפעה על המדדים השונים. כלומר, ככל שמספר הקשתות עולה, ערכי המדדים יישארו בקירוב קבועים.

סיכום

בפרויקט זה רכשנו ידע נרחב בתחום של קריפטוגרפיה בכלל, והוכחות באפס ידיעה בפרט.
למדנו לעומק כיצד מוגדרת מערכת הוכחה באפס ידיעה והתמקדנו בשני פרוטוקולים: ה-ZKP ל-3 צביעות של גרף וה-ZKP למעגל המילטוני בגרף.
תכננו את המימוש של הפרוטוקולים תוך בחינת חלופות שונות, ולבסוף החלטנו על הדרך המתאימה למימוש.
מימשנו את שני הפרוטוקולים בשפת פייתון, תוך שימוש בספריות שונות. לאחר מכן, הגדרנו ניסויים לבדיקת היעילות של הפרוטוקולים, הרצנו אותם ובהתאם לתוצאות הסקנו מסקנות בנוגע ליעילות התוכניות ולהשוואה ביניהן.
במהלך הפרויקט, בדקנו לעומק את הביצוע של כל שלב ונעזרנו במקורות מידע אמינים לצורך כך.

רעיון מעניין להמשך הוא בדיקת המימושים של הפרוטוקולים בהיבט של סקיילינג.
ניתן לבדוק מהי המגבלה של מחשב ממוצע בהרצת כל אחת מהתוכניות מבחינת גודל הקלט – הגרף המשותף (כלומר, לבדוק מהו גודל הקלט המקסימלי שעבורו התוכנית יכולה להתבצע על גבי מחשב ממוצע).
בהתאם לכך, ניתן להסיק באיזה סדר גודל של קלטים נדרש מעבר לשימוש בענן.
נשאיר זאת כשאלה פתוחה בפרויקט שלנו.

ביבליוגרפיה

Barker, E. (2016, January). *Recommendation for Key Management, Part 1: General.* Retrieved from NIST: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf

Blum, M. (1986). How to Prove a Theorem So No One Else Can Claim It [Blu86].

Fernàndez-València, R. (2021, November 30). *Commitment Schemes*. Retrieved from Medium: https://medium.com/iovlabs-innovation-stories/commitment-schemes-4f3590be8c5

Goldreich, O. Micali, S. Wigderson, A. (1986). Proofs that Yield Nothing But their Validity and a Methodology of Cryptographic Protocol Design [GMW86].

Goldreich, O. (2001). *Foundations of Cryptography - Volume 1*. Cambridge University.

Leskovec, J. (n.d.). *Stanford Large Network Dataset Collection*. Retrieved from Stanford Network Analysis Project: https://snap.stanford.edu/data/index.html

נספחים

נספח 1: קישור לגיט של הפרויקט ב-GitHub

https://github.com/BarDaabul/Zero-knowledge_proofs_and_cryptographic_applications_final_project.git

נספח 2: ElGamalCommitmentScheme.py

```
#  implementation of ElGamalCommitmentScheme which providing a
computationally hiding and statistically binding
#  Submitted by:
#  Bar Avraham Daabul
#  Nadav Yosef Zada

import random


# Calculate the gcd of a and b
def gcd(a, b):
    if a < b:
        return gcd(b, a)
    elif a % b == 0:
        return b
    else:
        return gcd(b, a % b)


# Generate the secret key (sk) and the public key (pk)
def generateKeys(q, g):
    secretKey = random.randint(pow(2, 224), q)
    while gcd(q, secretKey) != 1:
        secretKey = random.randint(pow(2, 224), q)  # secret key (sk)

    publickKey = modPow(g, secretKey, q)  # public key (pk) [g^key %
q]
    return secretKey, publickKey


# Calculate modular exponentiation
def modPow(a, b, c):
    x = 1
    y = a
    while b > 0:
        if b % 2 != 0:
            x = (x * y) % c
        y = (y * y) % c
        b = int(b / 2)
    return x % c


# Commit on value
def commit(q, g, value, y):
    r = random.randint(1, q-2)  # random value
    c1 = modPow(g, r, q)  # c1 = g^r % q
    c2 = (value * modPow(y, r, q)) % q  # c2 = [value * (y^r % q)] %
q
    C = [c1, c2]  # the commitment is C = [c1,c2]
    return C, r
```

```
# Verify on value (dec = [value, r])
def verify(q, g, y, dec, commitFromCommitStep):
    value = dec[0]
    r = dec[1]

    checkC1 = modPow(g, r, q)  # checkC1 = g^r % q
    checkC2 = (value * modPow(y, r, q)) % q  # checkC2 = [value *
(y^r % q)] % q

    if(checkC1 != commitFromCommitStep[0]) or (checkC2 !=
commitFromCommitStep[1]):
        return False  # the reveal for the commit is not
corresponding to the given values
    return True  # the reveal for the commit is correct
```

נספח 3: PrivateColoring.py

```
# Submitted by:
# Bar Avraham Daabul
# Nadav Yosef Zada

# coloring[i] is the color of the vertex i, each of the value 1,2,3
represents a different color
# In order to use a specific coloring, update the value of coloring
array to the required coloring
coloring = [1, 2, 3, 3]  # the coloring for the graph G
wrongColoring = [1, 1, 1, 2]  # a wrong coloring for G (if the
verifier will choose the edges (0,1) or (0,2), it will reject.

coloring2 = [2, 1, 2, 1, 2, 1, 3, 2, 3, 3]  # the coloring for the
graph G2
coloring3 = [2, 2, 1, 2, 2]  # the coloring for the graph G3 (not 3-
coloring), it will reject in probability of 5/8

coloringForFacebookGraph = [1 for i in range(4039)]  # A wrong
coloring for facebookGraph
coloringForP2PGnutellaGraph = [1 for i in range(10879)]  # A wrong
coloring for P2PGnutellaGraph

coloringForP2PGnutellaSubgraph = [1 for i in range(10000)]  # A wrong
coloring for P2PGnutellaSubgraph (changing for each subgraph)

# Return the private coloring (regarding to the required graph)
def getPrivateColoring():
    return coloring2  #
coloringForP2PGnutellaSubgraph#coloringForP2PGnutellaGraph#coloringFo
rFacebookGraph#coloring3#coloring2#wrongColoring#coloring
```

נספח 4: CommonGraph.py

הקוד הבא מתייחס לקלט המשותף (הגרף) ב-ZKP ל-3 צביעות של גרף:

```
# Submitted by:
# Bar Avraham Daabul
# Nadav Yosef Zada
import random  # it is used for the measurements

# The graph is represented by Adjacency Matrix

# In order to create a specific graph, run the function
createGraphByAdjMatrix and get the required graph.
# Afterwards, update the value of G to the required graph.

# A 3-coloring graph with 4 vertices and 3 edges
G = [[0, 1, 1, 0], [1, 0, 0, 1], [1, 0, 0, 0], [0, 1, 0, 0]]

# A 3-coloring graph with 10 vertices and 15 edges
G2 = [[0, 1, 0, 0, 0, 1, 0, 0, 0, 1], [1, 0, 1, 0, 0, 0, 0, 0, 1, 0],
[0, 1, 0, 1, 0, 0, 1, 0, 0, 0], [0, 0, 1, 0, 1, 0, 0, 0, 0, 1], [0,
0, 0, 1, 0, 1, 0, 0, 1, 0], [1, 0, 0, 0, 1, 0, 1, 0, 0, 0], [0, 0, 1,
0, 0, 1, 0, 1, 0, 0], [0, 0, 0, 0, 0, 0, 1, 0, 1, 1], [0, 1, 0, 0, 1,
0, 0, 1, 0, 0], [1, 0, 0, 1, 0, 0, 0, 1, 0, 0]]

# A graph with 5 vertices and 8 edges and which is not 3-coloring
G3 = [[0, 1, 0, 0, 1], [1, 0, 1, 1, 1], [0, 1, 0, 1, 1], [0, 1, 1, 0,
1], [1, 1, 1, 1, 0]]

# Printing the required graph
def createGraphByAdjMatrix():
    #  n is the number of vertices
    #  m is the number of edges
    print("Enter the number of vertices and the number of edges:")
    n, m = map(int, input().split())
    G = [[0 for i in range(n)]for j in range(n)]
    for i in range(m):
        print("Enter 2 vertices that will have an edge between
them:")
        u, v = map(int, input().split())
        G[u][v] = 1
        G[v][u] = 1
    print(G)


# A function that gets a path to graph and the index of the first
line in the dataset that represent a node, and it returns an
adjacency matrix for the given graph
def loadGraphAsAdjMatrix(pathToGraph, indexFirstLineOfData):
    numOfNodes = findNumOfNodes(pathToGraph, indexFirstLineOfData)
    graph = open(pathToGraph)
    lines = graph.readlines()[indexFirstLineOfData:]
    adjMatrix = [[0 for i in range(numOfNodes)]for j in
range(numOfNodes)]
    for line in lines:
        u, v = map(int, line.split())
        adjMatrix[u][v] = 1
        adjMatrix[v][u] = 1
    return adjMatrix
```

```
# A function that gets a path to graph and the index of the first
line in the dataset that represent a node, and it returns the number
of nodes in the given graph
def findNumOfNodes(pathToGraph, indexFirstLineOfData):
    graph = open(pathToGraph)
    lines = graph.readlines()[indexFirstLineOfData:]
    maxNode = -1
    for line in lines:
        u, v = map(int, line.split())
        maxNode=max(u, v, maxNode)
    numOfNodes = maxNode + 1  # the nodes are numbered from 0
    return numOfNodes


# Return the required graph
def getCommonGraph():
    # return G
    return G2
    # return G3

    # graphFacebook = loadGraphAsAdjMatrix("facebook_combined.txt",
0)  # A graph with 4039 nodes and 88234 edges
    # return graphFacebook

    # graphP2PGnutella = loadGraphAsAdjMatrix("p2p-Gnutella04.txt",
4)  # A graph with 10879 nodes and 39994 edges
    # return graphP2PGnutella

    # subgraphFacebook =
loadCheckingGraphAsAdjMatrix("facebook_combined_NumEdges_88000.txt")
# we ran the protocol for a lot of subgraphs for the measurements
(effect of the number of edges)
    # return subgraphFacebook

    # subgraphP2PGnutella = loadCheckingGraphAsAdjMatrix("p2p-
Gnutella04_NumNodes_10000.txt")  # we ran the protocol for a lot of
subgraphs for the measurements (effect of the number of nodes)
    # return subgraphP2PGnutella


# ***************************
# Functions for measurements:
# ***************************
# A function that gets a path to original graph, the index of the
first line in the dataset that represent a node, and the required num
of edges
# It creates a new sub-graph of the original graph with the required
number of edges
def createCheckingGraphByEdges(pathToOriginalGraph,
indexFirstLineOfData, reqNumEdges):
    graph = open(pathToOriginalGraph)
    lines = graph.readlines()[indexFirstLineOfData:]
    random_lines = random.sample(lines, reqNumEdges)
    name =
pathToOriginalGraph.split(".txt")[0]+"_NumEdges_"+str(reqNumEdges)
    numOfNodes = findNumOfNodes(pathToOriginalGraph,
indexFirstLineOfData)
    writeSubgraphToFile(random_lines, name, numOfNodes)
```

```
# A function that gets a path to original graph, the index of the
first line in the dataset that represent a node, and the required num
of nodes
# It creates a new sub-graph of the original graph with the required
number of nodes
def createCheckingGraphByNodes(pathToOriginalGraph,
indexFirstLineOfData, reqNumNodes):
    graph = open(pathToOriginalGraph)
    lines = graph.readlines()[indexFirstLineOfData:]
    reqLines = []
    for line in lines:
        u, v = map(int, line.split())
        if u < reqNumNodes and v < reqNumNodes:
            reqLines.append(line)
    name =
pathToOriginalGraph.split(".txt")[0]+"_NumNodes_"+str(reqNumNodes)
    writeSubgraphToFile(reqLines, name, reqNumNodes)


# A function that get a list of sub-graphs and the number of nodes
(n).
# It outputs to text file the value of n, the number of sub-graphs
(count) and the edges of each sub-graph
def writeSubgraphToFile(random_lines, name, numOfNodes):
    filename = f"{name}.txt"
    with open(filename, "w") as file:
        file.write(str(numOfNodes)+"\n")  # the first line will have
the number of nodes in the original graph
        for line in random_lines:
            file.write(line)
    return 0


# A function that gets a path to sub-graph, and it returns an
adjacency matrix for the given graph
def loadCheckingGraphAsAdjMatrix(pathToSubGraph):
    graph = open(pathToSubGraph)
    lines = graph.readlines()
    numOfNodes = lines[0]
    numOfNodes = int(numOfNodes)

    lines = lines[1:]  # indexFirstLineOfData=1
    adjMatrix = [[0 for i in range(numOfNodes)]for j in
range(numOfNodes)]
    for line in lines:
        u, v = map(int, line.split())
        adjMatrix[u][v] = 1
        adjMatrix[v][u] = 1
    return adjMatrix

#if __name__ == '__main__':
#    createGraphByAdjMatrix()

#    for i in range(11):
#        createCheckingGraphByEdges("facebook_combined.txt", 0,
(i+1)*8000)

#    for i in range(10):
#        createCheckingGraphByNodes("p2p-Gnutella04.txt", 4,
(i+1)*1000)
```

נספח 5: Prover.py

הקוד הבא מתייחס למוכיח ב-ZKP ל-3 צביעות של גרף:

```
# Submitted by:
# Bar Avraham Daabul
# Nadav Yosef Zada

# The Prover is the server and it communicates with the Verifier (the
client)

import socket
import threading
import random
import numpy as np
import CommonGraph  # it contains the common graph
import PrivateColoring  # it contains the private coloring of the
graph
import ElGamalCommitmentScheme  # it contains ElGamal commitment
scheme
import timeit  # it is used for the running time measurements
import os  # it is used for the memory measurements
import psutil  # it is used for the memory measurements
import sys  # it is used for the communication measurements

# Define constants
HOST = '127.0.0.1'  # Standard loopback IP address (localhost)
PORT = 60000  # Port to listen on (non-privileged ports are > 1023)
FORMAT = 'utf-8'  # Define the encoding format of messages from
client-server
ADDR = (HOST, PORT)  # Creating a tuple of IP+PORT
NORM_BUF_SIZE = 1024  # The buffer size for the normal (not
significantly heavy) data that is received in the communication


# Function that starts the server
def start_server():
    server_socket.bind(ADDR)  # binding socket with specified IP+PORT
tuple
    server_socket.listen(1)  # Server is open for connection to one
client

    while True:
        connection, address = server_socket.accept()  # Waiting for
client to connect to server (blocking call)

        if threading.activeCount() - 1 == 1:  # Check if the maximum
number of clients has reached
            connection.send("Error: The prover is already connected
to a verifier".encode(FORMAT))
            connection.close()
            continue

        if threading.activeCount() - 1 < 1:  # Check if the maximum
number of clients hasn't reached
            connection.send("Accepted".encode(FORMAT))
            thread = threading.Thread(target=handle_client,
args=(connection, address))  # Creating new Thread object.
            thread.start()  # Starting the new thread (<=> handling
new client)
```

```
# Function that handles a single client connection
# It performs the steps of the prover in the protocol
def handle_client(conn, addr):
    print("\nProver:\n")
    # values for using the ElGamal commitment scheme
    q = random.randint(pow(2, 224), pow(2, 256))  # order of the
cyclic group
    g = random.randint(2, q)  # generator of the group
    key, y = ElGamalCommitmentScheme.generateKeys(q, g)  # key =
secret key (sk), y = public key (pk) [y=g^key % q]

    publicValues = [q, g, y]  # public values for using the ElGamal
commitment scheme
#    print("publicValues:", publicValues)
    conn.send(str(publicValues).encode(FORMAT))
    conn.recv(NORM_BUF_SIZE).decode(FORMAT)  # wait until gets ack
("Public values were received") from the verfier

    # The measurements are tested from the beginning of the protocol
(when the participants get the protocol's inputs):
# $    startCurrTimeMs = timeit.default_timer()  # The start time of
the current section where the prover is running
    # for the total memory consumption measurements
# $    process = psutil.Process(os.getpid())  # get the current
process
# $    memory_start = process.memory_info().rss / (1024 * 1024)  #
get the memory usage (MB) in the beginning

    G = CommonGraph.getCommonGraph()  # the common graph
    coloring = PrivateColoring.getPrivateColoring()  # the private 3-
coloring

    # perform the first step of the prover
    print("Perform the first step of the prover")
    permColoring, commitmentsArr, randValForDecArr =
firstStepOfProver(conn, publicValues, coloring)
# $    totalCommunication =
sys.getsizeof(str(commitmentsArr).encode(FORMAT))  # for the
communication measurements
#    print("commitmentsArr:\n", commitmentsArr)
#    print("Prover's first step has finished\n")

# $    totalTimeMs = timeit.default_timer() - startCurrTimeMs  #
current run time measurement
    chosenEdgeStr = conn.recv(NORM_BUF_SIZE).decode(FORMAT)  # wait
until gets an edge from the verfier
# $    startCurrTimeMs = timeit.default_timer()  # The start time of
the current section where the prover is running
# $    totalCommunication +=
sys.getsizeof(chosenEdgeStr.encode(FORMAT))  # for the communication
measurements
    chosenEdge = eval(chosenEdgeStr)
    print("The verifier chose the edge: ", chosenEdge)

    # perform the second step of the prover
    print("Perform the second step of the prover")
    edgeDec = secondStepOfProver(conn, permColoring,
randValForDecArr, chosenEdge)
```

```
# $    totalCommunication +=
sys.getsizeof(str(edgeDec).encode(FORMAT))  # for the communication
measurements
#    print("Prover's second step has finished\n")

# $    totalTimeMs += (timeit.default_timer() - startCurrTimeMs)  #
current run time measurement
    resultFromVerifier = conn.recv(NORM_BUF_SIZE).decode(FORMAT)  #
wait until gets the result (accept/reject) from the verfier
# $    startCurrTimeMs = timeit.default_timer()  # The start time of
the current section where the prover is running
# $    totalCommunication +=
sys.getsizeof(resultFromVerifier.encode(FORMAT))  # for the
communication measurements

    if resultFromVerifier == "Accept":
        print("\nThe verifier accepted the claim!")
    else:
        print("\nThe verifier rejected the claim!")


    # The protocol finished its operation. From here, there are some
measurement results:
# $    print("\nmeasurement result:")
    # total run time measurement
# $    totalTimeMs += (timeit.default_timer() - startCurrTimeMs)  #
current run time measurement
# $    print(f"The execution time of the prover: {totalTimeMs:.3f}
sec")

    # total memory consumption measurement:
# $    memory_end = process.memory_info().rss / (1024 * 1024)  # get
the memory usage (MB) in the end
# $    memory_usage = memory_end - memory_start
# $    print(f"The memory usage of the prover: {memory_usage:.3f}
MB")

    # total communication measurement:
# $    print(f"The amount of data used for communication of both
sides (prover and verifier): {totalCommunication / (1024 * 1024):.3f}
MB")

    conn.close()  # close the connection with the verifier


# Function that performs the prover's first step (perform a
permutation of the colors and commits on the new colors)
def firstStepOfProver(conn, publicValues, coloring):
    colors = [1, 2, 3]  # the representation of the colors
    permColoring = PermForColoring(colors, coloring)  # perform a
permutation of the colors
    commitmentsArr, randValForDecArr = commitOnColoring(permColoring,
publicValues)  # get the commitments and the random values for the
decommitments
    conn.send(str(commitmentsArr).encode(FORMAT))  # send the
commitments to the verifier
    return permColoring, commitmentsArr, randValForDecArr


# Function that gets an array of 3-colors and an array of coloring.
```

```
# It performs permutation on the colors and returns an array of the
new coloring
def PermForColoring(colors, coloring):
    premColors = np.random.permutation(colors)
##    while premColors[0] == 1 and premColors[1] == 2 and
premColors[2] == 3:  # check if premColors is the same as colors
##        premColors = np.random.permutation(colors)

    permColoring = np.copy(coloring)
    for i in range(len(permColoring)):
        if coloring[i] == 1:
            permColoring[i] = premColors[0]
        if coloring[i] == 2:
            permColoring[i] = premColors[1]
        if coloring[i] == 3:
            permColoring[i] = premColors[2]
    return permColoring


# Function that gets a coloring of the vertices and public values
# It returns an array of commitments on the coloring and the random
values that was used in the commitments.
def commitOnColoring(permColoring, publicValues):
    commitmentsArr = [[0]*2 for i in range(len(permColoring))]  # it
will contain the commitments for the color of each vertex (each
commitment has 2 components)
    randValForDecArr = [0]*len(permColoring)  # it will contain the
random values that was used for the commitments

    q = publicValues[0]
    g = publicValues[1]
    y = publicValues[2]
    for i in range(len(permColoring)):  # commit on the colors of
each vertex
        commitmentsArr[i], randValForDecArr[i] =
ElGamalCommitmentScheme.commit(q, g, permColoring[i], y)
    return commitmentsArr, randValForDecArr


# Function that performs the prover's second step (send the dec for
the two vertices that was chosen by the verifier)
def secondStepOfProver(conn, permColoring, randValForDecArr,
chosenEdge):
    v1 = chosenEdge[0]  # the first node of the chosen edge
    v2 = chosenEdge[1]  # the second node of the chosen edge
    colorVal1 = permColoring[v1]  # the color (the value) of the
first node
    colorVal2 = permColoring[v2]  # the color (the value) of the
second node
    randomVal1 = randValForDecArr[v1]  # the random value of the
commitment on the first node
    randomVal2 = randValForDecArr[v2]  # the random value of the
commitment on the second node
    dec1 = [colorVal1, randomVal1]
    dec2 = [colorVal2, randomVal2]
    edgeDec = [dec1, dec2]  # the decs of the chosen edge
    conn.send(str(edgeDec).encode(FORMAT))  # send the decs of the
chosen edge to the verifier
    return edgeDec
```

```
# Main
if __name__ == '__main__':
    IP = socket.gethostbyname(socket.gethostname())  # finding the
current IP address
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Opening Server socket
    start_server()
```

הערה: הפקודות למדידת זמן הריצה, הזיכרון והתקשורת מופיעים בקוד כהערות עם הסימון הבא: **# $** להרצת התוכנית עם המדידות, יש להסיר את ההערות האלה.

נספח 6: Verifier.py

הקוד הבא מתייחס למוודאת ב-ZKP ל-3 צביעות של גרף:

```
# Submitted by:
# Bar Avraham Daabul
# Nadav Yosef Zada

# The Verifier is the client and it communicates with the Prover (the
server)
import random
import socket
import CommonGraph  # it contains the common graph
import ElGamalCommitmentScheme  # it contains ElGamal commitment
scheme
import timeit  # it is used for the running time measurements
import os  # it is used for the memory measurements
import psutil  # it is used for the memory measurements

# Define constants
HOST = '127.0.0.1'  # The server's hostname or IP address
PORT = 60000  # The port used by the server
FORMAT = 'utf-8'
ADDR = (HOST, PORT)  # Creating a tuple of IP+PORT
NORM_BUF_SIZE = 1024  # The buffer size for the normal (not
significantly heavy) data that is received in the communication
MAX_BUF_SIZE = 4194304  # the buffer size for the heavy data that is
received in the communication


# Function that starts the client
def start_client():
    print("\nVerifier:\n")
    client_socket.connect((HOST, PORT))  # Connecting to server's
socket
    isAccepted = client_socket.recv(NORM_BUF_SIZE).decode(FORMAT)  #
Receiving data from the server after trying to connect
    if isAccepted == "Error: The prover is already connected to a
verifier":
        print(isAccepted)
        return

    # Receiving public values for using the commitment scheme from
the server
    publicValuesStr =
client_socket.recv(NORM_BUF_SIZE).decode(FORMAT)
    publicValues = eval(publicValuesStr)
#    print("publicValues:", publicValues)
```

```
    client_socket.send("Public values were received".encode(FORMAT))
# Sending acknowledgment to the server

    # The measurements are tested from the beginning of the protocol
(when the participants get the protocol's inputs):
# $    startCurrTimeMs = timeit.default_timer()  # The start time of
the current section where the verifier is running
    # for the total memory consumption measurements
# $    process = psutil.Process(os.getpid())  # get the current
process
# $    memory_start = process.memory_info().rss / (1024 * 1024)  #
get the memory usage (MB) in the beginning

    G = CommonGraph.getCommonGraph()  # the common graph

# $    totalTimeMs = timeit.default_timer() - startCurrTimeMs  #
current run time measurement

    # Receiving an array of commitments on the colors of each vertex
from the server
    commitmentsArrStr =
client_socket.recv(MAX_BUF_SIZE).decode(FORMAT)
# $    startCurrTimeMs = timeit.default_timer()  # The start time of
the current section where the verifier is running
    commitmentsArr = convertStringToMatrix(commitmentsArrStr)
#    print("commitmentsArr\n", commitmentsArr)

    print("Perform the first step of the verifier")
    chosenEdge = firstStepOfVerifier(client_socket, G)  # choose
random edge and send it to prover, chosenEdge=[u, v]
#    print("Verifier's first step has finished\n")

# $    totalTimeMs += (timeit.default_timer() - startCurrTimeMs)  #
current run time measurement
    edgeDecStr = client_socket.recv(NORM_BUF_SIZE).decode(FORMAT)  #
wait until gets dec1 and dec2 from the prover (after prover's second
step)
# $    startCurrTimeMs = timeit.default_timer()  # The start time of
the current section where the verifier is running
    edgeDec = eval(edgeDecStr)
    print("Perform the second step of the verifier")
    is3Coloring = secondStepOfVerifier(client_socket, chosenEdge,
publicValues, edgeDec, commitmentsArr)
    if(is3Coloring):
        print("\nAccept the claim - the graph is a 3-coloring!")
    else:
        print("\nReject the claim - the graph is not a 3-coloring!")
#    print("\nVerifier's second step has finished\n")

    # The protocol finished its operation. From here, there are some
measurement results:
# $    print("\nmeasurement result:")
    # total run time measurement
# $    totalTimeMs += (timeit.default_timer() - startCurrTimeMs)  #
current run time measurement
# $    print(f"The execution time of the verifier: {totalTimeMs:.3f}
sec")

    # total memory consumption measurement:
# $    memory_end = process.memory_info().rss / (1024 * 1024)  # get
the memory usage (MB) in the end
```

```
# $    memory_usage = memory_end - memory_start
# $    print(f"The memory usage of the verifier: {memory_usage:.3f}
MB")

    return


# Function that convert string to an array (matrix form)
def convertStringToMatrix(matrix):
    newMatrix = matrix.replace("\n ", ", ")
    newMatrix = newMatrix.replace(" ", ", ")
    newMatrix = newMatrix.replace(",, ", ", ")
    return eval(newMatrix)


# Function that performs the verifier's first step (choose a random
edge from the graph and send it to the prover)
def firstStepOfVerifier(client_socket, G):
    # If the matrix length is nxn (the adjacency matrix is square),
so the vertices are numbered from 0 to n-1.
    # We will choose random edge from the graph.
    chosenEdge = chooseRandomEdge(G)  # the chosen edge
    print("The chosen edge: ", chosenEdge)
    client_socket.send(str(chosenEdge).encode(FORMAT))
    return chosenEdge


# Function that get a graph and return a random edge from the graph
def chooseRandomEdge(G):
    edges = []
    # Take all the edges from the adjacency matrix to a list of edges
    for u in range(len(G)):
        for v in range(u+1, len(G)):
            if G[u][v] == 1:
                edges.append([u, v])

    chosenEdge = random.choice(edges)  # choose a random edge
    return chosenEdge


# Function that performs the verifier's second step (check the
opening of the commitments and output accept or reject)
def secondStepOfVerifier(client_socket, chosenEdge, publicValues,
edgeDec, commitmentsArr):
    dec1 = edgeDec[0]  # the dec for the first vertex
    dec2 = edgeDec[1]  # the dec for the second vertex
    colorVal1 = dec1[0]  # the color (the value) for the first vertex
    colorVal2 = dec2[0]  # the color (the value) for the second
vertex

    # the public values
    q = publicValues[0]
    g = publicValues[1]
    y = publicValues[2]

    v1 = chosenEdge[0]  # the index of the first vertex
    v2 = chosenEdge[1]  # the index of the second vertex

# if the opening of the commitments is correct and the colors of the
vertices are different, is3Coloring = true
# else (one of the tests gives false), is3Coloring = false
```

```
    is3Coloring = ElGamalCommitmentScheme.verify(q, g, y, dec1,
commitmentsArr[v1]) and ElGamalCommitmentScheme.verify(q, g, y, dec2,
commitmentsArr[v2]) and (colorVal1 != colorVal2)
    if(is3Coloring):
        client_socket.send(str("Accept").encode(FORMAT))
    else:
        client_socket.send(str("Reject").encode(FORMAT))
    return is3Coloring


# Main
if __name__ == "__main__":
    IP = socket.gethostbyname(socket.gethostname())  # finding the
current IP address
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Opening Client socket
    start_client()
    client_socket.close()  # Closing client's connection with server
(<=> closing socket)
    print("\nThe interaction with the prover has finished")
```

<u>הערה:</u> הפקודות למדידת זמן הריצה, הזיכרון והתקשורת מופיעים בקוד כהערות עם הסימון הבא: **$ #** להרצת התוכנית עם המדידות, יש להסיר את ההערות האלה.

נספח 7: PrivateHamiltonianCycle.py

```
# Submitted by:
# Bar Avraham Daabul
# Nadav Yosef Zada

# hamiltonianCycle has the visited nodes in the order of the required
path
# In order to use a specific hamiltonian cycle, update the value of
the array to the required path
hamiltonianCycle = [0, 3, 1, 2, 0]  # a hamiltonian cycle in M
wrongPath = [0, 3, 1, 0]  # a wrong path for M (not hamiltonian
cycle)

hamiltonianCycle2 = [0, 1, 2, 9, 8, 7, 6, 5, 14, 13, 12, 19, 15, 16,
17, 18, 10, 11, 3, 4, 0]  # an hamiltonian cycle in M2
hamiltonianCycle3 = [0, 2, 3, 4, 2]  # a wrong path for M3 (M3
doesn't have a hamiltonian cycle)

# A path for reducedFacebookGraph
pathForFacebookGraph = [i for i in range(1000)]  # It is 400 for
facebook_combined_for_avg_weight_metrics
pathForFacebookGraph.append(0)

# A path for P2PGnutellaSubgraph (changing for each subgraph)
pathForP2PGnutellaSubgraph = [i for i in range(1500)]
pathForP2PGnutellaSubgraph.append(0)


# Return the private hamiltonian cycle (regarding to the required
graph)
def getPrivateHamiltonianCycle():
    return hamiltonianCycle2  #
pathForP2PGnutellaSubgraph#pathForFacebookGraph#hamiltonianCycle3#ham
iltonianCycle2#wrongPath#hamiltonianCycle
```

נספח 8: CommonGraph.py

הקוד הבא מתייחס לקלט המשותף (הגרף) ב-ZKP למעגל המילטוני בגרף:

```
# Submitted by:
# Bar Avraham Daabul
# Nadav Yosef Zada
import random  # it is used for the measurements

# The graph is represented by Adjacency Matrix

# In order to create a specific graph, run the function
createDirectedGraphByAdjMatrix and get the required graph.
# Afterwards, update the value of M to the required graph.

# A directed graph with hamiltonian cycle which has 4 vertices and 5
edges
M = [[0, 0, 0, 1], [1, 0, 1, 0], [1, 0, 0, 0], [0, 1, 0, 0]]

# A directed graph with hamiltonian cycle which has 20 vertices and
60 edges
M2 = [[0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 1,
0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 1, 0,
1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0], [1, 0, 0, 1, 0, 0,
0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1, 0, 1, 0, 0,
0, 0, 0, 0, 0, 0, 1, 0, 0, 0], [0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0,
0, 0, 0, 1, 0, 0], [0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0,
1, 0], [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1], [0, 0,
0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0], [0, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1], [0, 0, 0, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 0, 1, 0, 1, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 0, 0, 0, 0, 1, 0, 1], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 1, 0, 0, 1, 0]]

# A directed graph with no hamiltonian cycle which has 5 vertices and
6 edges
M3 = [[0, 1, 1, 0, 0], [0, 0, 0, 0, 0], [0, 1, 0, 1, 0], [0, 0, 0, 0,
1], [0, 0, 1, 0, 0]]


# Printing the required graph
def createDirectedGraphByAdjMatrix():
    #  n is the number of vertices
    #  m is the number of edges
    print("Enter the number of vertices and the number of edges:")
    n, m = map(int, input().split())
    G = [[0 for i in range(n)]for j in range(n)]
    for i in range(m):
        print("Enter 2 vertices (u and v) so that there will be an
edge from u to v:")
        u, v = map(int, input().split())
        G[u][v] = 1
    print(G)
```

```
# A function that gets a path to directed graph and the index of the
first line in the dataset that represent a node, and it returns an
adjacency matrix for the given graph
def loadDirectedGraphAsAdjMatrix(pathToGraph, indexFirstLineOfData):
    numOfNodes = findNumOfNodes(pathToGraph, indexFirstLineOfData)
    graph = open(pathToGraph)
    lines = graph.readlines()[indexFirstLineOfData:]
    adjMatrix = [[0 for i in range(numOfNodes)]for j in
range(numOfNodes)]
    for line in lines:
        u, v = map(int, line.split())
        adjMatrix[u][v] = 1
    return adjMatrix


# A function that gets a path to graph and the index of the first
line in the dataset that represent a node, and it returns the number
of nodes in the given graph
def findNumOfNodes(pathToGraph, indexFirstLineOfData):
    graph = open(pathToGraph)
    lines = graph.readlines()[indexFirstLineOfData:]
    maxNode = -1
    for line in lines:
        u, v = map(int, line.split())
        maxNode=max(u, v, maxNode)
    numOfNodes = maxNode + 1  # the nodes are numbered from 0
    return numOfNodes


# Return the required graph
def getCommonGraph():
    # return M
    return M2
    # return M3

    # subgraphFacebook =
loadCheckingDirectedGraphAsAdjMatrix("facebook_combined_reduced_NumEd
ges_9000.txt")  # we ran the protocol for a lot of subgraphs for the
measurements (effect of the number of edges)
    # return subgraphFacebook

    # subgraphP2PGnutella =
loadCheckingDirectedGraphAsAdjMatrix("p2p-
Gnutella04_NumNodes_1500.txt")  # we ran the protocol for a lot of
subgraphs for the measurements (effect of the number of nodes)
    # return subgraphP2PGnutella


# ***************************
# Functions for measurements:
# ***************************
# A function that gets a path to original graph, the index of the
first line in the dataset that represent a node, and the required num
of edges
# It creates a new sub-graph of the original graph with the required
number of edges
def createCheckingGraphByEdges(pathToOriginalGraph,
indexFirstLineOfData, reqNumEdges):
    graph = open(pathToOriginalGraph)
    lines = graph.readlines()[indexFirstLineOfData:]
```

```
    random_lines = random.sample(lines, reqNumEdges)
    name =
pathToOriginalGraph.split(".txt")[0]+"_NumEdges_"+str(reqNumEdges)
    numOfNodes = findNumOfNodes(pathToOriginalGraph,
indexFirstLineOfData)
    writeSubgraphToFile(random_lines, name, numOfNodes)


# A function that gets a path to original graph, the index of the
first line in the dataset that represent a node, and the required num
of nodes
# It creates a new sub-graph of the original graph with the required
number of nodes
def createCheckingGraphByNodes(pathToOriginalGraph,
indexFirstLineOfData, reqNumNodes):
    graph = open(pathToOriginalGraph)
    lines = graph.readlines()[indexFirstLineOfData:]
    reqLines = []
    for line in lines:
        u, v = map(int, line.split())
        if u < reqNumNodes and v < reqNumNodes:
            reqLines.append(line)
    name =
pathToOriginalGraph.split(".txt")[0]+"_NumNodes_"+str(reqNumNodes)
    writeSubgraphToFile(reqLines, name, reqNumNodes)


# A function that get a list of sub-graphs and the number of nodes
(n).
# It outputs to text file the value of n, the number of sub-graphs
(count) and the edges of each sub-graph
def writeSubgraphToFile(random_lines, name, numOfNodes):
    filename = f"{name}.txt"
    with open(filename, "w") as file:
        file.write(str(numOfNodes)+"\n")  # the first line will have
the number of nodes in the original graph
        for line in random_lines:
            file.write(line)
    return 0


# A function that gets a path to sub-graph, and it returns an
adjacency matrix for the given graph
def loadCheckingDirectedGraphAsAdjMatrix(pathToSubGraph):
    graph = open(pathToSubGraph)
    lines = graph.readlines()
    numOfNodes = lines[0]
    numOfNodes = int(numOfNodes)

    lines = lines[1:]  # indexFirstLineOfData=1
    adjMatrix = [[0 for i in range(numOfNodes)]for j in
range(numOfNodes)]
    for line in lines:
        u, v = map(int, line.split())
        adjMatrix[u][v] = 1
    return adjMatrix


# if __name__ == '__main__':

#    createDirectedGraphByAdjMatrix()
```

```
#
*********************************************************************
****************************************
# graphs for finding the metrics as a function of the number of nodes
and the number of edges (for bit=0 and bit=1 separately)

#    # a reduced graph of facebook with 1000 nodes (it has 9890
edges), that will use for checking the effect of the number of edges.
The file of the reduced graph: facebook_combined_reduced.txt
#    createCheckingGraphByNodes("facebook_combined.txt", 0, 1000)

#    for i in range(9):
#        createCheckingGraphByEdges("facebook_combined_reduced.txt",
1, (i+1)*1000)

#    for i in range(10):
#        createCheckingGraphByNodes("p2p-Gnutella04.txt", 4,
(i+1)*150)

#
*********************************************************************
*************************************
# graphs for finding the average weighted metrics as a function of
the number of nodes and the number of edges

#    # a reduced graph of facebook with 400 nodes (it has 3062
edges), that will use for checking the effect of the number of edges.
The file of the reduced graph:
facebook_combined_for_avg_weight_metrics.txt
#    createCheckingGraphByNodes("facebook_combined.txt", 0, 400)

#    for i in range(6):
#
createCheckingGraphByEdges("facebook_combined_for_avg_weight_metrics.
txt", 1, (i+1)*500)

#    for i in range(6):
#        createCheckingGraphByNodes("p2p-Gnutella04.txt", 4,
(i+1)*100)
```

נספח 9: Prover.py

הקוד הבא מתייחס למוכיח ב-ZKP למעגל המילטוני בגרף:

```
# Submitted by:
# Bar Avraham Daabul
# Nadav Yosef Zada

# The Prover is the server and it communicates with the Verifier (the
client)

import socket
import threading
import random
import numpy as np
import CommonGraph  # it contains the common graph
import PrivateHamiltonianCycle  # it contains the private hamiltonian
cycle of the graph
```

```
import ElGamalCommitmentScheme  # it contains ElGamal commitment
scheme
import timeit  # it is used for the running time measurements
import os  # it is used for the memory measurements
import psutil  # it is used for the memory measurements
import sys  # it is used for the communication measurements

# Define constants
HOST = '127.0.0.1'  # Standard loopback IP address (localhost)
PORT = 60000  # Port to listen on (non-privileged ports are > 1023)
FORMAT = 'utf-8'  # Define the encoding format of messages from
client-server
ADDR = (HOST, PORT)  # Creating a tuple of IP+PORT
NORM_BUF_SIZE = 1024  # The buffer size for the normal (not
significantly heavy) data that is received in the communication


# Function that starts the server
def start_server():
    server_socket.bind(ADDR)  # binding socket with specified IP+PORT
tuple
    server_socket.listen(1)  # Server is open for connection to one
client

    while True:
        connection, address = server_socket.accept()  # Waiting for
client to connect to server (blocking call)

        if threading.activeCount() - 1 == 1:  # Check if the maximum
number of clients has reached
            connection.send("Error: The prover is already connected
to a verifier".encode(FORMAT))
            connection.close()
            continue

        if threading.activeCount() - 1 < 1:  # Check if the maximum
number of clients hasn't reached
            connection.send("Accepted".encode(FORMAT))
            thread = threading.Thread(target=handle_client,
args=(connection, address))  # Creating new Thread object.
            thread.start()  # Starting the new thread (<=> handling
new client)


# Function that handles a single client connection
# It performs the steps of the prover in the protocol
def handle_client(conn, addr):
    print("\nProver:\n")
    # values for using the ElGamal commitment scheme
    q = random.randint(pow(2, 224), pow(2, 256))  # order of the
cyclic group
    g = random.randint(2, q)  # generator of the group
    key, y = ElGamalCommitmentScheme.generateKeys(q, g)  # key =
secret key (sk), y = public key (pk) [y=g^key % q]

    publicValues = [q, g, y]  # public values for using the ElGamal
commitment scheme
#    print("publicValues:", publicValues)
    conn.send(str(publicValues).encode(FORMAT))
    conn.recv(NORM_BUF_SIZE).decode(FORMAT)  # wait until gets ack
("Public values were received") from the verfier
```

```
    # The measurements are tested from the beginning of the protocol
(when the participants get the protocol's inputs):
# $    startCurrTimeMs = timeit.default_timer()  # The start time of
the current section where the prover is running
    # for the total memory consumption measurements
# $    process = psutil.Process(os.getpid())  # get the current
process
# $    memory_start = process.memory_info().rss / (1024 * 1024)  #
get the memory usage (MB) in the beginning

    M = CommonGraph.getCommonGraph()  # the common graph
    hamiltonianCycle =
PrivateHamiltonianCycle.getPrivateHamiltonianCycle()  # the private
hamiltonian cycle

    # perform the first step of the prover
    print("Perform the first step of the prover")
    newM, permNodes, commitmentsArr, randValForDecArr =
firstStepOfProver(conn, publicValues, M)
# $    totalCommunication =
sys.getsizeof(str(commitmentsArr).encode(FORMAT))  # for the
communication measurements
#    print("commitmentsArr:\n", commitmentsArr)
#    print("Prover's first step has finished\n")

# $    totalTimeMs = timeit.default_timer() - startCurrTimeMs  #
current run time measurement
    chosenBitStr = conn.recv(NORM_BUF_SIZE).decode(FORMAT)  # wait
until gets a bit from the verfier
# $    startCurrTimeMs = timeit.default_timer()  # The start time of
the current section where the prover is running
# $    totalCommunication +=
sys.getsizeof(chosenBitStr.encode(FORMAT))  # for the communication
measurements
    chosenBit = eval(chosenBitStr)
    print("The verifier chose the bit: ", chosenBit)

    # perform the second step of the prover
    print("Perform the second step of the prover:")
    strDec = secondStepOfProver(conn, newM, permNodes,
randValForDecArr, chosenBit, hamiltonianCycle)
# $    totalCommunication += sys.getsizeof(strDec.encode(FORMAT))  #
for the communication measurements
#    print("Prover's second step has finished\n")

# $    totalTimeMs += (timeit.default_timer() - startCurrTimeMs)  #
current run time measurement
    resultFromVerifier = conn.recv(NORM_BUF_SIZE).decode(FORMAT)  #
wait until gets the result (accept/reject) from the verfier
# $    startCurrTimeMs = timeit.default_timer()  # The start time of
the current section where the prover is running
# $    totalCommunication +=
sys.getsizeof(resultFromVerifier.encode(FORMAT))  # for the
communication measurements

    if resultFromVerifier == "Accept":
        print("\nThe verifier accepted the claim!")
    else:
        print("\nThe verifier rejected the claim!")
```

```
    # The protocol finished its operation. From here, there are some
measurement results:
# $    print("\nmeasurement result:")
    # total run time measurement
# $    totalTimeMs += (timeit.default_timer() - startCurrTimeMs)  #
current run time measurement
# $    print(f"The execution time of the prover: {totalTimeMs:.3f}
sec")

    # total memory consumption measurement:
# $    memory_end = process.memory_info().rss / (1024 * 1024)  # get
the memory usage (MB) in the end
# $    memory_usage = memory_end - memory_start
# $    print(f"The memory usage of the prover: {memory_usage:.3f}
MB")

    # total communication measurement:
# $    print(f"The amount of data used for communication of both
sides (prover and verifier): {totalCommunication / (1024 * 1024):.3f}
MB")

    conn.close()  # close the connection with the verifier


# Function that performs the prover's first step
# It performs a permutation of the nodes and get a new graph (adj
matrix) and commits on the entries of the new graph
def firstStepOfProver(conn, publicValues, M):
    numOfNodes = len(M)
    currNodes = [i for i in range(numOfNodes)]  # the representation
of the nodes

    newM, permNodes = PermForNodes(M, currNodes)  # perform a
permutation of the nodes and get a new graph

    commitmentsArr, randValForDecArr = commitOnNewMatrix(newM,
publicValues)  # get the commitments and the random values for the
decs
    conn.send(str(commitmentsArr).encode(FORMAT))  # send the
commitments to the verifier
    return newM, permNodes, commitmentsArr, randValForDecArr


# Function that gets a graph and an array of nodes
# It performs a permutation on the nodes and returns the new graph
(adj matrix)
def PermForNodes(M, currNodes):
    numOfNodes = len(M)
    permNodes = np.random.permutation(currNodes)
    permNodes = [node for node in permNodes]  # changing the form of
representation of the permutation

    newM = [[0 for i in range(numOfNodes)] for j in
range(numOfNodes)]
    for i in range(numOfNodes):
        for j in range(numOfNodes):
            permI = permNodes[i]
            permJ = permNodes[j]
            newM[permI][permJ] = M[i][j]
    return newM, permNodes
```

```
# Function that gets a new graph (adj matrix) and public values
# It returns an array of commitments on the entries of the matrix and
the random values that was used in the commitments
def commitOnNewMatrix(newM, publicValues):
    numOfNodes = len(newM)
    commitmentsArr = [[0]*2 for i in range(numOfNodes*numOfNodes)]  #
it will contain the commitments for the entries of the matrix (each
commitment has 2 components)
    randValForDecArr = [0]*numOfNodes*numOfNodes  # it will contain
the random values that was used for the commitments

    q = publicValues[0]
    g = publicValues[1]
    y = publicValues[2]
    for k in range(numOfNodes*numOfNodes):  # commit on the value of
each entry
        row = int(k / numOfNodes)
        col = int(k % numOfNodes)
        commitmentsArr[k], randValForDecArr[k] =
ElGamalCommitmentScheme.commit(q, g, newM[row][col], y)
    return commitmentsArr, randValForDecArr


# Function that performs the prover's second step
# If chosenBit = 0, it sends to the verifier the permutation on the
nodes and the decs of the new graph
# If chosenBit = 1, it sends to the verifier the decs of the
hamiltonian cycle's edges in the new graph
def secondStepOfProver(conn, newM, permNodes, randValForDecArr,
chosenBit, hamiltonianCycle):
    if(chosenBit == 0):
        print("Reveal the permutation and the commitments of the new
graph's entries")
        return revealPermAndNewM(conn, newM, permNodes,
randValForDecArr)  # sends to the verifier the permutation on the
nodes and the decs of the new graph (newM)
    else:  # chosenBit == 1
        print("Reveal the commitments of the hamiltonian cycle's
edges in the new graph")
        return revealNewHamiltonianCycle(conn, newM, permNodes,
randValForDecArr, hamiltonianCycle)  # sends to the verifier the decs
of the hamiltonian cycle's edges in the new graph (newM)


# A Function that sends to the verifier the permutation of the nodes
and the decs of the new graph (newM)
def revealPermAndNewM(conn, newM, permNodes, randValForDecArr):
    strDec = str(permNodes)+"&"
    numOfNodes = len(newM)
    decArr = [[0]*2 for i in range(numOfNodes*numOfNodes)]  # it will
contain the decs (value and random value) of each entry in the new
graph (newM)
    for k in range(numOfNodes * numOfNodes):  # decommit of the
values of each entry
        row = int(k / numOfNodes)
        col = int(k % numOfNodes)
        decArr[k] = [newM[row][col], randValForDecArr[k]]
    strDec += str(decArr)
```

```
    conn.send(strDec.encode(FORMAT))  # send the permutation of the
nodes and the decs of the new graph (newM)
    return strDec


# A Function that sends to the verifier the decs of the hamiltonian
cycle's edges in the new graph (newM)
def revealNewHamiltonianCycle(conn, newM, permNodes,
randValForDecArr, hamiltonianCycle):
    numOfNodes = len(newM)
    permHamiltonianCycle = [permNodes[i] for i in hamiltonianCycle]
    strDec = str(permHamiltonianCycle) + "&"

    decArr = [[0] * 2 for i in range(len(permHamiltonianCycle)-1)]  #
it will contain the decs (value and random value) of the entries in
newM that represents the new hamiltonian cycle
    for k in range(len(decArr)):  # decommit of the required values
        row = permHamiltonianCycle[k]
        col = permHamiltonianCycle[k+1]
        decArr[k] = [newM[row][col], randValForDecArr[row *
numOfNodes + col]]
    strDec += str(decArr)
    conn.send(strDec.encode(FORMAT))  # send the decs of the
hamiltonian cycle's edges in the new graph (newM)
    return strDec


# Main
if __name__ == '__main__':
    IP = socket.gethostbyname(socket.gethostname())  # finding the
current IP address
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Opening Server socket
    start_server()
```

הערה: הפקודות למדידת זמן הריצה, הזיכרון והתקשורת מופיעים בקוד כהערות עם הסימון הבא: **# $**
להרצת התוכנית עם המדידות, יש להסיר את ההערות האלה.

נספח 10: Verifier.py

הקוד הבא מתייחס למוודאת ב-ZKP למעגל המילטוני בגרף:

```
# Submitted by:
# Bar Avraham Daabul
# Nadav Yosef Zada

# The Verifier is the client and it communicates with the Prover (the
server)
import random
import socket
import CommonGraph  # it contains the common graph
import ElGamalCommitmentScheme  # it contains ElGamal commitment
scheme
import timeit  # it is used for the running time measurements
import os  # it is used for the memory measurements
import psutil  # it is used for the memory measurements

# Define constants
HOST = '127.0.0.1'  # The server's hostname or IP address
```

```
PORT = 60000  # The port used by the server
FORMAT = 'utf-8'
ADDR = (HOST, PORT)  # Creating a tuple of IP+PORT
NORM_BUF_SIZE = 1024  # The buffer size for the normal (not
significantly heavy) data that is received in the communication
MAX_BUF_SIZE = 268435456  # the buffer size for the heavy data that
is received in the communication

# Function that starts the client
def start_client():
    print("\nVerifier:\n")
    client_socket.connect((HOST, PORT))  # Connecting to server's
socket
    isAccepted = client_socket.recv(NORM_BUF_SIZE).decode(FORMAT)  #
Receiving data from the server after trying to connect
    if isAccepted == "Error: The prover is already connected to a
verifier":
        print(isAccepted)
        return

    # Receiving public values for using the commitment scheme from
the server
    publicValuesStr =
client_socket.recv(NORM_BUF_SIZE).decode(FORMAT)
    publicValues = eval(publicValuesStr)
#    print("publicValues:", publicValues)
    client_socket.send("Public values were received".encode(FORMAT))
# Sending acknowledgment to the server

    # The measurements are tested from the beginning of the protocol
(when the participants get the protocol's inputs):
# $    startCurrTimeMs = timeit.default_timer()  # The start time of
the current section where the verifier is running
    # for the total memory consumption measurements
# $    process = psutil.Process(os.getpid())  # get the current
process
# $    memory_start = process.memory_info().rss / (1024 * 1024)  #
get the memory usage (MB) in the beginning

    M = CommonGraph.getCommonGraph()  # the common graph

# $    totalTimeMs = timeit.default_timer() - startCurrTimeMs  #
current run time measurement

    # Receiving an array of commitments on the entries of the new
graph (newM)
    commitmentsArrStr =
client_socket.recv(MAX_BUF_SIZE).decode(FORMAT)
# $    startCurrTimeMs = timeit.default_timer()  # The start time of
the current section where the verifier is running
    commitmentsArr = convertStringToMatrix(commitmentsArrStr)
#    print("commitmentsArr\n", commitmentsArr)

    print("Perform the first step of the verifier")
    chosenBit = firstStepOfVerifier(client_socket)  # choose a random
bit and send it to prover
#    print("Verifier's first step has finished\n")

# $    totalTimeMs += (timeit.default_timer() - startCurrTimeMs)  #
current run time measurement
```

```
    strDec = client_socket.recv(MAX_BUF_SIZE).decode(FORMAT)  # wait
until gets dec from the prover (after prover's second step)
# $    startCurrTimeMs = timeit.default_timer()  # The start time of
the current section where the verifier is running
    dec = [strDec.split("&")[0], strDec.split("&")[1]]
    dec1 = eval(dec[0])
    dec2 = convertStringToMatrix(dec[1])
    # if b=0, dec=[permNodes, decArr (decs for all the entries in
newM)].
    # if b=1, dec=[permHamiltonianCycle, decArr (decs of the entries
that represent the new hamiltonian cycle)].
    dec = [dec1, dec2]

    print("Perform the second step of the verifier:")
    hasHamiltonianCycle = secondStepOfVerifier(client_socket,
chosenBit, publicValues, dec, commitmentsArr, M)
    if hasHamiltonianCycle:
        print("\nAccept the claim - the graph has a hamiltonian
cycle!")
    else:
        print("\nReject the claim - the graph doesn't have a
hamiltonian cycle!")
#    print("\nVerifier's second step has finished\n")

    # The protocol finished its operation. From here, there are some
measurement results:
# $    print("\nmeasurement result:")
    # total run time measurement
# $    totalTimeMs += (timeit.default_timer() - startCurrTimeMs)  #
current run time measurement
# $    print(f"The execution time of the verifier: {totalTimeMs:.3f}
sec")

    # total memory consumption measurement:
# $    memory_end = process.memory_info().rss / (1024 * 1024)  # get
the memory usage (MB) in the end
# $    memory_usage = memory_end - memory_start
# $    print(f"The memory usage of the verifier: {memory_usage:.3f}
MB")

    return


# Function that convert string to an array (matrix form)
def convertStringToMatrix(matrix):
    newMatrix = matrix.replace("\n ", ", ")
    newMatrix = newMatrix.replace(" ", ", ")
    newMatrix = newMatrix.replace(",, ", ", ")
    return eval(newMatrix)


# Function that performs the verifier's first step (choose a random
bit and send it to the prover)
def firstStepOfVerifier(client_socket):
    chosenBit = random.randint(0, 1)  # choose random bit (0 or 1)
    print("The chosen bit: ", chosenBit)
    client_socket.send(str(chosenBit).encode(FORMAT))
    return chosenBit


# Function that performs the verifier's second step
```

```
def secondStepOfVerifier(client_socket, chosenBit, publicValues, dec,
commitmentsArr, M):
    if chosenBit == 0:
        print("Check if the decommits are valid and if the new graph
is isomorphic to the original graph")
        hasHamiltonianCycle =
checkDecAndIsomorphicGraphs(publicValues, dec, commitmentsArr, M)
    else:  # chosenBit == 1
        print("Check if the decommits are valid and if the given path
is a hamiltonian cycle")
        hasHamiltonianCycle =
checkDecAndHamiltonianCycle(publicValues, dec, commitmentsArr,
len(M))

    if (hasHamiltonianCycle):
        client_socket.send(str("Accept").encode(FORMAT))
    else:
        client_socket.send(str("Reject").encode(FORMAT))

    return hasHamiltonianCycle


# A Function that check the decs of the new graph and check that the
new graph is isomorphic to the original graph
def checkDecAndIsomorphicGraphs(publicValues, dec, commitmentsArr,
M):
    # dec=[permNodes, decArr (decs for all the entries in newM)].
    permNodes = dec[0]
    decArr = dec[1]  # decs for all the entries in newM

    # the public values
    q = publicValues[0]
    g = publicValues[1]
    y = publicValues[2]

    numOfNodes = len(M)
    newM = [[0 for i in range(numOfNodes)] for j in
range(numOfNodes)]
    for k in range(numOfNodes * numOfNodes):  # numOfNodes *
numOfNodes = len(decArr)
        isValidDec = ElGamalCommitmentScheme.verify(q, g, y,
decArr[k], commitmentsArr[k])
        if(not isValidDec):
            return False
        currValEntry = decArr[k][0]
        if(currValEntry == 1):  # Otherwise the value is initialized
to 0
            row = int(k / numOfNodes)
            col = int(k % numOfNodes)
            newM[row][col] = currValEntry

    # create the mapping for the nodes
    mappingNodes = {}
    for i in range(len(permNodes)):
        mappingNodes[i] = permNodes[i]
    return areIsomorphicGraphs(M, newM, mappingNodes)


# A Function that get an original graph, a new graph and a mapping of
the permutation of the nodes
```

```
# It returns True if the graphs are isomorphic by the given mapping,
and False otherwise
def areIsomorphicGraphs(M, newM, mappingNodes):
    if len(M) != len(newM):  # check if the number of nodes is the
same
        return False
    # check that every entry in the original graph has the same value
as the corresponding entry in the new graph using the permutation
    for i in range(len(M)):
        for j in range(len(M)):
            if M[i][j] !=
newM[mappingNodes.get(i)][mappingNodes.get(j)]:
                return False
    return True


# A Function that check the decs of the new cycle and check that the
new cycle is hamiltonian
def checkDecAndHamiltonianCycle(publicValues, dec, commitmentsArr,
numOfNodes):
    # dec=[permHamiltonianCycle, decArr (decs of the entries that
represent the new hamiltonian cycle)].
    permHamiltonianCycle = dec[0]
    decArr = dec[1]  # decs of the entries that represent the new
hamiltonian cycle

    # the public values
    q = publicValues[0]
    g = publicValues[1]
    y = publicValues[2]

    for k in range(len(permHamiltonianCycle)-1):  #
len(permHamiltonianCycle)-1 = len(decArr)
        row = permHamiltonianCycle[k]
        col = permHamiltonianCycle[k+1]

        isValidDec = ElGamalCommitmentScheme.verify(q, g, y,
decArr[k], commitmentsArr[row * numOfNodes + col])
        currValEntry = decArr[k][0]
        if (not isValidDec) or (currValEntry != 1):  # if the dec is
not valid or there is no edge in the current entry, return False
            return False
    return isHamiltonianCycle(permHamiltonianCycle, numOfNodes)


# A Function that get a path in a graph, and the num of nodes in the
graph.
# It returns True if the path is hamiltonian cycle in the graph, and
False otherwise
def isHamiltonianCycle(permHamiltonianCycle, numOfNodes):
    # checking if permHamiltonianCycle is a cycle (starts and ends in
same node)
    if (permHamiltonianCycle[0] !=
permHamiltonianCycle[len(permHamiltonianCycle)-1]):
        return False

    startNode = permHamiltonianCycle[0]

    # count the number of appearance for each node
    countNodesAppearance = [0]*numOfNodes
    for node in permHamiltonianCycle:
```

```
        countNodesAppearance[node] += 1

    if (countNodesAppearance[startNode] != 2):  # if the start node
doesn't appear exactly twice, return False
        return False
    for i in range(numOfNodes):
        if (i != startNode and countNodesAppearance[i] != 1):  # if
the current node is not the start node and it doesn't appear exactly
one time, return False
            return False
    return True


# Main
if __name__ == "__main__":
    IP = socket.gethostbyname(socket.gethostname())  # finding the
current IP address
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Opening Client socket
    start_client()
    client_socket.close()  # Closing client's connection with server
(<=> closing socket)
    print("\nThe interaction with the prover has finished")
```

<u>הערה:</u> הפקודות למדידת זמן הריצה, הזיכרון והתקשורת מופיעים בקוד כהערות עם הסימון הבא: **# $** להרצת התוכנית עם המדידות, יש להסיר את ההערות האלה.

<u>נספח 11: טבלאות עם ערכי המדדים שהתקבלו מההרצות עבור חישוב המדדים המשוקללים הממוצעים</u>

ערכי המדדים עבור ההרצות לבדיקת ההשפעה של מספר הצמתים:

<u>ריצה 1</u>

מספר הצמתים (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
100	5.013	0.077	0.818	2.324	3.293
200	19.639	0.332	3.234	10.113	10.375
300	43.681	43.797	14.169	37.203	49.621
400	79.371	1.247	12.873	38.094	39.316
500	121.255	2.06	20.018	58.816	60.691
600	174.28	2.989	28.71	85.004	86.824

<u>ריצה 2</u>

מספר הצמתים (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
100	5.051	4.809	1.588	4.23	6.109
200	19.197	0.321	3.226	10.133	10.371
300	43.021	0.73	7.241	21.461	22.227
400	76.541	77.624	25.57	63.871	88.594
500	126.906	129.754	40.121	99.906	138.832
600	178.775	182.692	57.655	146.196	199.781

<u>ריצה 3</u>

מספר הצמתים (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
100	5.126	0.084	0.818	3.117	3.305
200	19.7	20.032	6.39	16.89	22.531
300	43.366	0.711	7.237	21.434	22.043
400	76.946	78.126	25.549	63.977	88.566
500	121.108	1.964	19.991	58.871	60.668
600	175.652	2.899	28.55	84.986	86.676

ריצה 4

מספר הצמתים (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
100	5.011	0.079	0.817	3.214	3.297
200	19.905	20.378	6.42	16.652	22.578
300	42.631	43.184	14.365	36.212	49.918
400	78.867	1.245	12.897	37.551	88.59
500	120.18	120.87	39.939	99.793	137.066
600	174.954	2.861	28.908	85.012	87.035

ריצה 5

מספר הצמתים (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
100	5.08	4.832	1.599	4.223	6.277
200	19.285	19.73	6.383	16.3	22.52
300	43.787	44.658	14.43	36.422	50.02
400	75.294	76.675	25.559	63.867	88.59
500	125.894	127.972	40.054	100.043	138.758
600	178.057	182.039	57.627	145.082	199.898

ריצה 6

מספר הצמתים (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
100	5.226	0.107	0.817	3.129	3.305
200	19.272	0.312	3.232	9.301	10.379
300	44.67	0.75	7.232	21.867	22.219
400	75.994	77.668	25.632	63.992	88.691
500	120.417	2.03	20.073	58.848	60.535
600	182.974	2.943	28.841	84.977	86.969

ערכי המדדים עבור ההרצות לבדיקת ההשפעה של מספר הקשתות:

ריצה 1

מספר הקשתות (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
500	77.481	77.763	25.612	63.797	88.641
1000	77.001	1.276	12.899	38.223	39.359
1500	80.168	1.288	12.942	37.645	39.125
2000	75.315	76.328	25.719	64.058	88.898
2500	78.439	81.51	25.7	63.922	88.926
3000	77.93	1.344	12.995	38.363	39.605

ריצה 2

מספר הקשתות (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
500	76.725	1.251	12.822	38.148	39.281
1000	78.327	1.264	12.884	38.289	39.34
1500	76.665	1.268	12.959	37.52	39.148
2000	79.537	79.203	25.664	63.996	88.797
2500	77.167	78.63	25.77	64.09	89.023
3000	78.626	80.394	25.806	64.122	88.997

ריצה 3

מספר הקשתות (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
500	78.383	79.733	25.621	64.73	88.707
1000	76.654	77.338	25.664	63.91	88.762
1500	79.026	1.258	12.916	37.754	39.09
2000	77.584	1.295	12.977	38.316	39.941
2500	79.79	1.313	12.999	37.852	39.551
3000	77.597	1.279	13.036	37.742	39.656

ריצה 4

מספר הקשתות (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
500	79.114	80.2	25.609	63.894	88.723
1000	78.484	1.266	12.895	38.137	39.348
1500	73.68	75.31	25.097	63.617	87.953
2000	75.571	77.264	25.612	64	88.766
2500	76.742	78.019	25.513	63.531	88.637
3000	78.318	79.554	25.796	64.902	88.043

ריצה 5

מספר הקשתות (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
500	78.162	1.273	12.879	37.527	39.305
1000	77.427	1.245	12.903	38.285	39.359
1500	78.213	79.615	25.646	63.903	88.785
2000	78.961	80.108	25.771	64.039	89
2500	77.849	1.278	13.035	38.266	39.566
3000	77.589	79.958	25.791	64.222	89.074

ריצה 6

מספר הקשתות (#)	זמן הריצה של המוכיח (s)	זמן הריצה של המוודאת (s)	מס' הביטים שעוברים בתקשורת (MB)	צריכת הזיכרון של המוכיח (MB)	צריכת הזיכרון של המוודאת (MB)
500	78.988	80.366	25.632	63.882	88.668
1000	76.545	77.831	25.642	63.746	88.73
1500	79.116	1.268	12.936	37.801	39.125
2000	83.3	1.278	12.963	38.266	39.578
2500	76.947	78.803	25.781	64.043	89.039
3000	76.926	1.29	13.044	38.324	39.656