עבודת גמר - תורת האינפורמציה 88-782

מגיש: רותם גזית, ת"ז 318446697

סמסטר חורף 2020

תוכן העניינים

1 הקדמה

מטרת הפרויקט הייתה לממש שיטת דחיסת נתונים ללא איבוד מידע (LOSSLESS COMPRESSION) שתגיע להישג טוב יותר מ ZIP.
במהלך הפיתוח ניסיתי מספר לא קטן של שיטות, וחלקן באמת הצליחו לפתור את האתגר, אותן אציג כאן.
בסוף הפרויקט אפרט בקצרה על שיטות נוספות שנבדקו לפתרון הבעיה.

1. שיטת Lempel Ziv Welch (LZW) לדחיסה וקידוד הנתונים
2. שיטת Lempel Ziv Welch לדחיסת הנתונים, יחד עם שיטת Huffman לקידוד הנתונים
3. שיטת Huffman לדחיסת הקובץ, כאשר הדחיסה מתבצעת ברמת המילה (ולא ברמת התו הבודד).

במסמך זה אציג תחילה את כל אחת מהשיטות, אציג את המימוש שלי עבורן, ולבסוף אציג השוואת ביצועים (יחס הדחיסה ביחס לפרמטרים שונים) בינהן ואל מול ZIP.

1.1 הנדסת תוכנה

הפרויקט פותח בשפת Python3.7 ואיננו מתבסס על ספריות חיצוניות (שאינן מובנות בתוך השפה).
הפרויקט נבנה באופן כזה שאוכל לבדוק בכל רגע נתון שהשיטות שפותחו אכן מצליחות לדחוס ולפרוש קבצים, תוך שמירה על מודולאריות שתאפשר לי להחליף לוגיקה בכל רגע נתון.
הוגדר Interface המגדיר שיטת דחיסה - ICompressor[1]. כלל השיטות הן מימוש של אותו הממשק. במקביל הוגדרו בדיקות unittests החלות באופן גלובלי על כל שיטת דחיסה המוודאות שפריסה של קובץ דחוס מחזירה את הקובץ המקורי[2]. לכל אחת משיטות הדחיסה הוגדרו בנוסף בדיקות משל עצמו[3].
שיטת הדחיסה הסופית שאני מציג כאן נקראת "Rotem Compression"[4]. למעשה מדובר במחלקה היורשת מ ICompressor ומחילה ברצף מספר שיטות דחיסה בזו אחר זו (ופורשת אותם בסדר ההפוך). ניתן להחליף את השיטה המוצגת בשעת ההפעלה ע"י החלפת המערך של שיטות הדחיסה[5].
תחת התיקייה data_models[6] נמצאות שתי מחלקות שמשמשות את את שיטות הדחיסה: BitStack[7] מאפשר התייחסות לביטים עצמם המרכיבים מחרוזת (למשל, לקחת את n הביטים הראשונים מתוך המחרוזת, או לדחוף מספר למחרוזת בתור קידוד רישא שלו כמספר טבעי), ו Node[8] מאפשר לבנות עצים בינאריים.
הקובץ app.py[9] הוא הקובץ הראשי של התכנית, הוא מקבל כקלט מהמשתמש בתור Arguments את שם הקובץ לקלט ולפלט התכנית, והוא מבצע דחיסה או פריסה עם שיטת RotemCompressor. הקובץ rgcompress[10] מאפשר לבצע את הפקודות ישירות ב command line באופן הבא:

for compression: `rgcompress -i <inputFile> -o <outputFile>`
for decompression: `rgcompress -d -i <inputFile> -o <outputFile>`

2 שיטות שנוסו

2.1 שיטת LZW

שיטת Lempel Ziv Welch הוצגה בשנת 1984 בתור הרחבה לשיטת LZ78 ע"י טרי וולץ'. השיטה מציגה שיפור ביצועים משמעותי לעומת LZ78 המקורי. כמו LZ78, היא מרכיבה מילון ע"ס הצירופים שכבר נראו בדרך שאציג בהמשך.
השינוי העיקרי ששיטה זו מציגה הוא בהגבלת גודל המילון (לרוב מגבילים ל 12 ביט). אך שמירת כלל המידע הדחוס ע"י יצוג בקודים באורך 12 ביט אינו דבר אידיאלי, ולכן השיטה מאפשרת לשמור את הקודים ב**אורכים משתנים**, כלומר להגדיל את מספר הביטים בהם מיוצג המידע רק ברגע הנכון.
אציג תחילה את האלגוריתם בצורתו הבסיסית:
נניח ויש לנו מחרוזת $S = \{s_n\}_{n=1}^{N}$ מעל א"ב מגודל k . נקבע את הגודל המקסימלי של המילון להיות $m > k$. תהי $\bar{S}$ המחרוזת הדחוסה שאנחנו רוצים לבנות.

---

[1] rotem_compressor/contract/ICompressor.py
[2] rotem_compressor/unittests/compression_testcase.py
[3] rotem_compressor/unittests
[4] rotem_compressor/rotem_compressor.py
[5] Reference to the 'compressions' list inside the class. Note the other methods are in comments.
[6] rotem_compressor/data_models
[7] rotem_compressor/data_models/bit_stack.py
[8] rotem_compressor/data_models/tree_node.py
[9] app.py
[10] rgcompress

1. אתחל מילון D בגודל k המכיל את כל התווים בא"ב.
תהי הפונקציה $D(\{q_i\})$ המחזירה את מיקומה של המחרוזת $\{q_i\}$ בתוך המילון D. מתקיים $s_i \in D$ לכל $1 \leq i \leq N$.

2. בשלב ה i, קח את j המקסימלי כך שהמחרוזת $s_{n_i} \dots s_{n_i+j} \in D$. נוסיף את $\bar{s}_i = D(s_{n_i} \dots s_{n_i+j})$ למחרוזת הדחוסה שלנו.

3. אם $|D| < m$, הוסף את $s_{n_i} \dots s_{n_i+j+1}$ (שאינה קיימת במילון) לתוך המילון D וקבע $D(s_{n_i} \dots s_{n_i+j+1}) = |D|$ כאשר $|D|$ כמות התווים במילון לאחר ההוספה.

4. קבע $n_{i+1} = s_{n_i+j+1}$. אם $n_i + j + 1 \leq N$, חזור לשלב 2.

נותר כעת להבין כיצד נייצג את המחרוזות מהא"ב המוגדל שבנינו (למעשה אנחנו הרחבנו את הא"ב להיות בגודל m). נניח שכל תו s_i במחרוזת המקורית הוא בית (2^8 ביטים). אזי הגודל הראשוני של D יהיה 2^8. על כן, כל תו בD שנוסיף בשלב 3, בהכרח ידרוש יותר מ 2^8 ביטים. בעקבות כך, אינטואיטיבית היינו בוחרים לייצג כל תו ב $\bar{S}$ ע"י $2^{\lceil log(m) \rceil}$ ביטים.
שיטה זו מאוד בזבזנית, לכל תו בא"ב המקורי יהיו הרבה אפסים מובילים.
שיטה "זולה" יותר תהיה לייצג את $\bar{s}_i$ ע"י $2^{\lceil log(|D|) \rceil}$, כלומר ע"י גודל המילון ברגע חישוב $\bar{s}_i$. כיוון שאנחנו יודעים שכל תו במחרוזת $\bar{S}$ מייצג הגדלה של המילון (שכן ייצגנו את תת המחרוזת הארוכה ביותר שיכלנו באותו רגע), זה שקול ל $2^{\lceil log(\min\{k+i,\ m\}) \rceil}$ תווים עבור התו $\bar{s}_i$.
באופן זה, כמות התווים הדרושה לייצוג גדלה ככל שהמילון גדל, וזה כמובן טוב יותר מקביעת הגודל מקסימלי לכל המילים כבר ברגע הראשון.

לאור אלו, ניתן לפרוס את המחרוזת ע"י:

1. אתחול מילון D

2. קריאת $2^{\lceil log(\min\{k+i,\ m\}) \rceil}$ ביטים בשלב ה i כדי לקבל את התו $\bar{s}_i$

3. המרה $D^{-1}(\bar{s}_i)$ לערך שמאחורי הקוד ושרשור למחרוזת הפרוסה (*)

4. אם $|D| < m$, הוסף למילון D את המחרוזת המורכבת מ $D^{-1}(\bar{s}_{i-1})D^{-1}(\bar{s}_i)_1$ (כלומר המחרוזת הקודמת שתרגמנו יחד עם התו הראשון מהמחרוזת החדשה)

(*) במידה ו $D^{-1}(\bar{s}_i)$ לא קיים - זה אומר שהקוד המקורי הוא $D^{-1}(\bar{s}_{i-1})D^{-1}(\bar{s}_{i-1})_1$, כלומר קוד שבזמן הקידוד נוצר ומיד נעשה בו שימוש - למשל המחרוזת aaa תקודד בתור [a, aa], וכשנגיע לתו השני בפרישת המחרוזת - הקוד aa עוד לא קיים.

אינטואיטיבית ברור שככל שנאפשר ל m להיות גדול יותר, מחרוזות יותר ארוכות יידחסו לתו בודד, ומצד שני ידרשו יותר ביטים לייצוג תווים החל משלב כלשהו, זאת כמובן על חשבון כמות זיכרון גדולה יותר הדרושה בזמן הדחיסה והפריסה. כיוון שאנחנו מייצגים כל תו בלכל היותר $2^{\lceil log(m) \rceil}$ תווים, נבחר ערכי m מהצורה 2^p (אחרת אנחנו סתם "מפסידים" מחרוזות שיכלנו לייצג באותה העלות).
שיטות נוספות לשיפור התהליך מאפשרות "איפוס" של המילון למצבו ההתחלתי במהלך הדרך, זאת על מנת לזהות באופן טוב יותר תבניות המשתנות בקובץ. שיפור כמו זה לא הכנסתי למימוש שלי.
המימוש של LZW נמצא בקובץ rotem_compressor/lzw.py. התהליך יודע לרוץ בשני אופנים - עם הפעלת הקידוד באורך המשתנה כפי שהוצג לעיל, או בלעדיו (ואז מוחזר מערך של מספרים מסוג int מגודל קבוע של 2^{32}). ניתן לבחור ערכי m שונים, ובפרק 3 אציג את השפעתו על איכות הדחיסה.

2.2 שיטת LZW + Huffman

השיטה השנייה אומרת את הדבר הבא: במקום להתעסק באופן קידוד התוצאות של LZW חזרה לבינארי, תוך שמירה על חיסכון גדול ככל האפשר - למה שלא ניתן לאלגוריתם אחר לעשות את העבודה ולמצוא את הקידוד היעיל ביותר.
את שיטת Huffman למדנו בכיתה בהרחבה: מתוך המידע שלנו, בונים עץ בינארי על סמך כמות החזרות של כל תו בא"ב שלנו במידע - ככל שתו תדיר יותר, כך הוא יופיע גבוה יותר בעץ (כלומר - המסלול מהשורש אליו קצר יותר). לאחר מכן, בונים קוד רישא ע"י קביעת קוד לכל עלה בעץ (כלומר - כל תו) ע"י המסלול אליו מהשורש. השיטה מבטיחה שיווצר לנו קוד רישא (מה שמאפשר קריאה שלו תו אחרי תו ועדיין לפרש כל קוד בצורה ייחודית).
כעת, ניתן לשלב אותה יחד עם LZW - אם ניקח מחרוזת מעל א"ב מגודל k, LZW יתן דחיסה שלו לא"ב מגודל $m > k$. כעת נפעיל על המחרוזת הדחוסה את אלגוריתם Huffman כדי לתת לכל תו בא"ב החדש שלנו ייצוג בקוד רישא על סמך תדירותו במידע הדחוס. אם נשמור את התוצאה הזו - למעשה התגברנו על הבעיה של הגדלת הא"ב שמתבצעת באלגוריתם הקודם.
בסופו של דבר קיבלנו מאלגוריתם האפמן עץ הבנוי לפי השכיחויות של כל אחד מהתווים במידע, ואת המידע כאשר כל תו מוחלף בייצוג שלו לפי האפמן. כעת צריך לקודד את העץ על מנת שנוכל לפרוס את הקובץ בחזרה - נעשה זאת בעזרת DFS עליו ונשמור את התוצאה יחד עם המידע הדחוס. כשנרצה לפרוס, נתרגם בחזרה את העץ מתוך ה DFS עליו, ואז לבנות תרגום מקוד לפי האפמן לתו המקורי. כמובן שככל שא"ב שלנו יהיה יותר גדול - העץ שלנו יהיה גדול יותר, ונצטרך יותר מחרוזת ארוכה יותר כדי לייצג אותו.

על מנת לייצג מספרים שלמים - אני מממש שיטה לייצוגם ע"י קוד רישא באופן הבא:

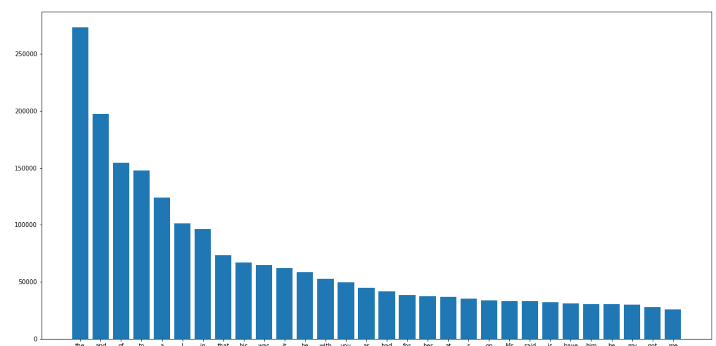
$$n \longrightarrow \underbrace{111}_{\lceil log(\lceil log(n+1)+1\rceil)\rceil \text{times}} \quad 0 \quad \underbrace{110}_{\lceil log(n+1)\rceil \text{ in binary}} \quad \underbrace{101101}_{n \text{ in binary}}$$

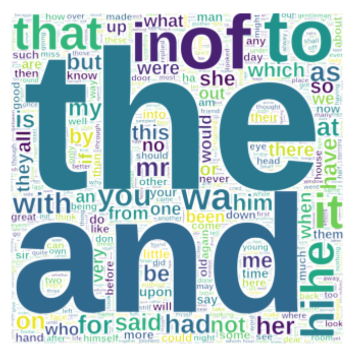
שיטה זו מאפשרת לי לייצג כל מספר $n \in \mathbb{N}$ ב $\lceil log(n+1)\rceil + 1 + 2\lceil log(\lceil log(n+1)\rceil + 1)\rceil$ תווים.
על מנת ליישם את השיטה נדרשתי ליישם תחילה את אלגוריתם האפמן. המימוש נמצא כאן: rotem_compressor/huffman_compression, והוא מחולק למספר תתי מחלקות. על מנת להשתמש בו לאחר LZW, צריך לשרשר אותם בזה אחר זה, כפי שקורה בrotem_compressor.py.
את המידע שיוצא מהאפמן אני שומר באופן הבא:

$h_1...h_{\tilde{N}}$	$d_1...d_t$	t	N
המידע המקודד האפמן	ה DFS על העץ מיוצג בקידוד טבעיים	מספר התווים ב DFS של העץ מיוצג בקידוד טבעיים	מספר התווים במידע המקורי מיוצג בקידוד טבעיים

2.3 שיטת Huffman לקידוד מילים

שתי השיטות המוצגות בסעיפים 2.1, 2.2 הן כלליות לחלוטין ועובדות על כל קובץ בינארי. אך במשימה הזו נדרשים לדחוס קובץ טקסטואלי, ועל כן ניתן להשתמש בעובדה הזו כדי להנות מיתרונות שיש למידע מסוג זה.
הרעיון של השיטה הזו הוא לנצל את העובדה שמדובר בקובץ טקסט. כיוון שזהו המצב, ניתן להניח שיש מילים שחוזרות הרבה יותר פעמים מהאחרות. התרשים הימני מציג word cloud של הקובץ - ככל שהמילה גדולה יותר - כך היא מופיעה מספר רב יותר של פעמים. משמאל אפשר לראות את כמות הפעמים שמופיעה כל מילה, כאשר מסתכלים על 30 המילים שמופיעות הכי הרבה פעמים.





כיוון שזהו המצב - היינו רוצים למצוא קוד שישמור למשל את המילה "little" (שמופיעה 12884 פעמים) במספר ביטים קטן ככל האפשר, לעומת המילה "Hooper" (שמופיעה רק פעמיים), למרות ששתיהן באותו האורך במחרוזת המקורית.
בסופו של דבר, לאחר הסרת רווחים, ירידות שורה וסימני פיסוק - ישנן בקובץ 57520 מילים ייחודיות (ורק 45296 אם מתעלמים מאותיות גדולות/קטנות) - כלומר פחות מ 2^{16} מילים. אם נמיר כל מילה או סימן פיסוק במספר בין 1 ל 2^{16} - נוכל להפעיל בקלות האפמן ולקבל דחיסה של הקובץ ביחס למילים שלו. לאחר מכן, כל שנותר זה לכתוב פעם אחת את כל המילים לפי סדר המספרים המייצגים אותן. כלומר האלגוריתם הוא:

1. מצא את כל המילים היחודיות במחרוזת (ללא תוי פיסוק, רווחים או ירידות שורה), וסדר אותם בסדר כלשהו. כתוב את כמות המילים, ולאחריה את המילים לפי הסדר, מופרדות ברווחים, במחרוזת הסופית.

2. עבור כל מילה או תו פיסוק במחרוזת - החלף אותה במספר המייצג את מיקומה ברשימת המילים.

3. הפעל אלגוריתם האפמן על רצף המספרים שהתקבל, כאשר גודל הא"ב הוא ככמות המילים. הוסף את התוצאה של האפמן למחרוזת הסופית.

כדי לפרוס את הדחיסה, יש לקרוא תחילה את רשימת המילים, לאחריה - לפרוס את קוד ההאפמן, ולהמיר כל מספר בתוצאה במילה שהיא מייצגת.
ייעול משמעותי שאפשר לעשות הוא לדחוס את רשימת המילים בנפרד - ע"י האפמן או שיטה אחרת, זאת כדי לחסוך את הכתיבה המלאה שלהן בתחילת התהליך. אני ביצעתי זאת עם LZW.
הקוד של השיטה הזו נמצא בקובץ rotem_compressor/words_encoder.py. החלק המורכב ביותר בו[11] הוא החלק של החלפת המילים והסימנים באינדקס המייצג אותן.

[11] __split_text_to_words_and_delimiters

השיטה הזו מאוד פשוטה אך עם זאת נתנה תוצאות טובות באופן שהפתיע אותי מאוד, והצליחה לדחוס את הקובץ טוב יותר מ ZIP.

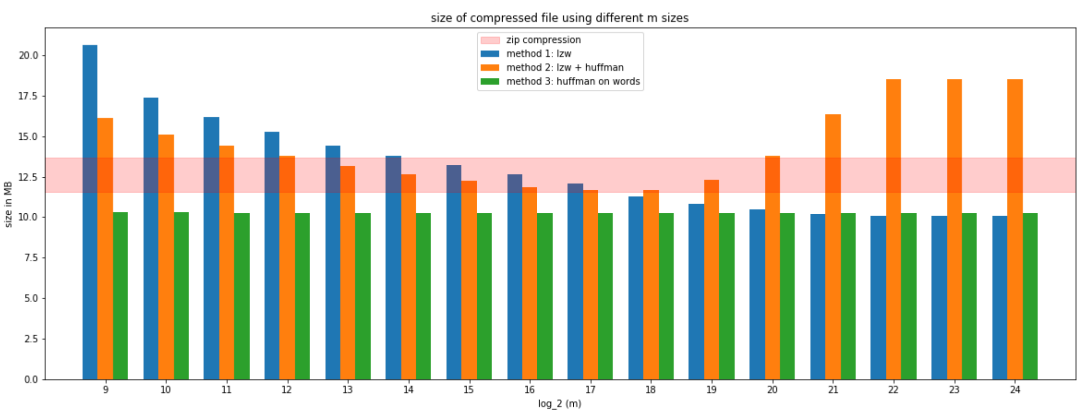
3 השוואה בין השיטות

אתחיל בכך ששלושת השיטות הצליחו להגיע (עם פרמטרים מסוימים) לתוצאות טובות משמעותית מול ZIP - ואף הגיעו להפרשים של 2MB ואף יותר. עם זאת - צריך להיות עדינים ולהבין מתי זה קרה: האם רק כשהקובץ היה "ארוך מספיק"? האם רק כשגודל המילון ש LZW יצר היה ענקי?

נשווה תחילה בין השיטות בפרמטרים שונים של m - גודל הא"ב המקסימלי ש LZW החזיק בכל אחד מהן. נזכיר -

1. השיטה הראשונה משתמשת ב LZW כולל קידוד באורכים משתנים
2. השיטה השנייה מפעילה LZW ואת התוצאות מעבירה דרך האפמן
3. השיטה השלישית מייצרת רשימת מילים ייחודיות בטקסט, מקודדת עם האפמן את הטקסט שבו החלפנו מילים באינדקסים שלהן, ומקודדת עם LZW את רשימת המילים.

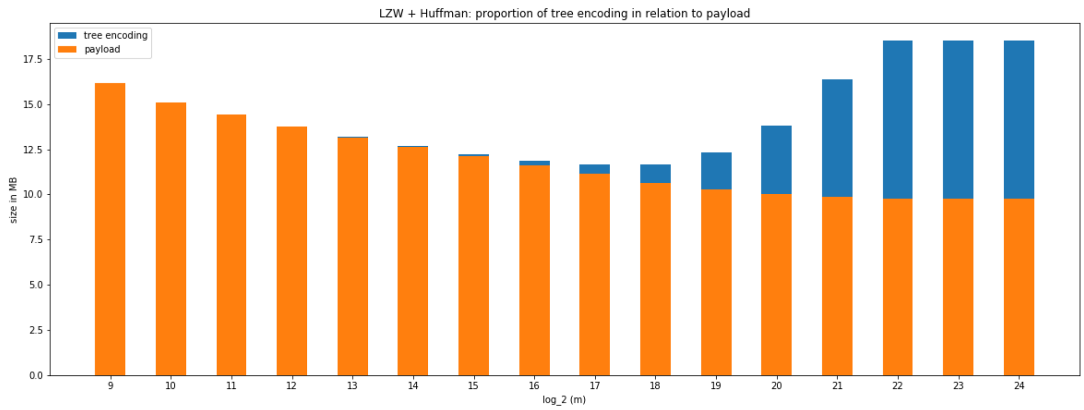
התרשים הבא מציג את גדלי הקבצים בכל אחת מהשיטות, כתלות בm גודל המילון. ציר x מציג $log_2(m)$ וציר y מציג את גודל הקובץ ב MB.
הטווח הצבוע באדום הוא טווח הגדלים שקובץ ZIP מחזיר (ניתן לבחור את איכות הדחיסה כמעט בכל תכנה)



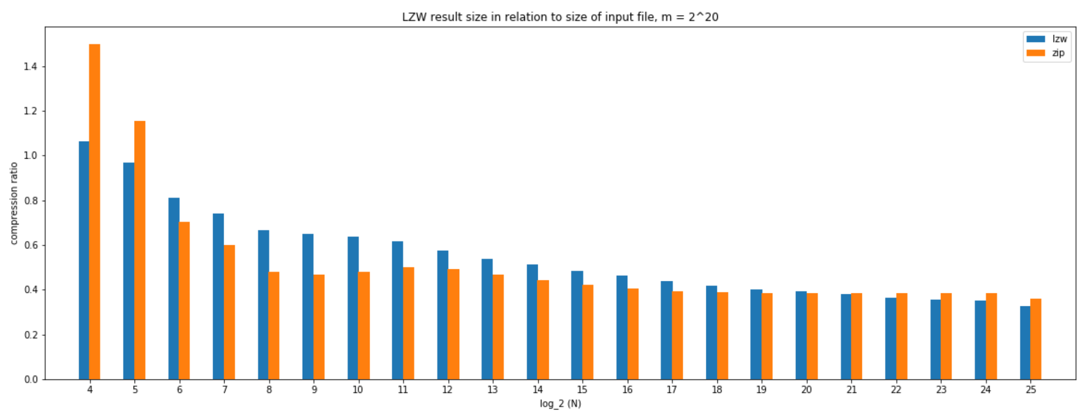
ניתן ללמוד מספר דברים מעניינים מהנתונים:

1. השיטה הטובה ביותר שנבדקה היא LZW עם קידוד באורכים משתנים, כאשר גודל המילון גדול מ2^{20}.
2. שיטת קידוד מילים כמעט ולא מושפעת מגודל המילון, זאת בגלל שהטקסט שמקודד עם LZW הוא ככל הנראה קצר מאוד (רק 50 אלף מילים)
3. שיטת LZW עם Huffman נפגעת כשגודל המילון גדול מידי - ככל הנראה בגלל שגודל העץ הופך להיות גדול מאוד ולכן הקידוד שלו תופס נפח רב.

כדי לאשש את הטענה השלישית, נחזור על הניסוי שוב, אבל הפעם נשמור את התוצאה של Huffman ללא קידוד העץ (כמובן שזו דחיסה שלא ניתנת לשחזור), כדי שנוכל להבין מה הנפח היחסי שתופס המידע ביחס לעץ. התרשים הבא מציג את גודל הקובץ בשיטה מספר 2, בחלוקה למידע עצמו ולעץ הקידוד שלו.



אפשר לראות שכשהמילון גדול מידי - חצי מהביטים בתוצאה הדחוסה הם הייצוג של הקידוד של העץ שלו.
כעת נתמקד ב LZW. ניקח $m = 2^{20}$, ונבחן **מתי** אנחנו מתחילים לראות הפרש בינו לבין ZIP. כלומר, לאחר כמה תווים מתוך הקובץ שלנו נוצר הפער. נסמן את מספר התווים שאנחנו דוחסים בכל פעם ב N. ציר y מציג את יחס הדחיסה ($\frac{\text{compressed}}{N}$).



מהתרשים אפשר לראות שרק כאשר דחסנו יותר מ 2^{21} בתים מתחיל להיות הפרש בגודל הקובץ לטובת LZW.

4 סיכום

בסופו של דבר, השיטה שאותה בחרתי לשימוש הינה שיטת LZW עם קידוד אורכים משתנה, כאשר קבעתי את גודל המילון להיות $m = 2^{20}$.

4.1 שיטות נוספות שנוסו בזמן העבודה

במהלך הדרך, ניסיתי להעזר בשיטות נוספות. הקוד שלהם הוסר מה git כדי למנוע בלבול (הוא נמצא בהיסטוריית ה commits). השיטות שניסיתי:

1. הוספת Run Length Encoding לאחר שימוש בשיטה 2 - האינטואיציה שלי אמרה שככל שהעץ יהיה "עמוק" יותר כך ב DFS על העץ יהיו רצפים ארוכים יותר של אפסים. לשם כך רציתי להוסיף RLE בתור דחיסה על גבי האפמן, אך הסתבר שזה לא הועיל בכלל היות והרצפים לא היו מספיק ארוכים כדי של RLE יהיה יתרון.

2. ניסיון לשחזר את שיטת bzip2 - שיטה זו משתמשת בהאפמן בתור המנוע העיקרי אך לפני זה מבצעת טרנספורמציות על המידע כך שיווצרו רצפים ארוכים מאותו תו ועליהם מפעילה RLE. היא משתמשת בשתי טרנספורמציות מקדימות:

   (א) התמרת בורווס-וילר (Burrows–Wheeler transform) - למעשה בוחרת פרמוטציה על בלוקים מתוך המידע שבהם ישנם רצפים חוזרים גדולים יותר מהטקסט המקורי

(ב) התמרת Move-to-front שמסתכלת על הטקסט כ"מחסנית" ובכל פעם מחליפה את התו באינדקס שלו במחסנית ושמה אותו בתחילתה.
כך רצפים כמו ababa מקודדים להיות $0, 1, 1, 1, 1$

לאחר שתי הטרנספורמציות היא מפעילה RLE על המידע כך שהפוטנציאל לדחוס יהיה גדול יותר. לאחר RLE מפעילים האפמן.
השיטה הזו - למרות שמימשתי את כולה - לא נתנה תוצאות שמתקרבות לתוצאות שהראתי כאן, גם מבחינת זמן הריצה.