פרוייקט סיום – דחיסת נתונים: LZSS

מגישים:

אבו רביע ראמי, 314820135,

איתמר אביר, 208273169.

מבוא:

אלגוריתם זה הוא הרחבה של האלגוריתם LZ77 (נרשום 77) עם יתרון ברור.

בעוד שאלגוריתם 77, במקרים מסוימים: מתקבל **קובץ קטן**, או שהוגדר ח**לון קטן** מדי שלא יתפוס חזרות (נזכיר דחיסה זו מבוססת מילון), ובעיקר **אנטרופיה גבוהה** – כלומר אין רצף חזרתי של אותיות, הקובץ שאנחנו לבסוף נקבל לאחר הדחיסה הוא קובץ בעל שטח אחסנה גדול יותר, מכיוון שבעת זיהוי רצף מתבצעות הפעולות הבאות: נכתוב היכן התחיל הרצף ומהו אורך הרצף, ובנוסף את התיו שנקרא.

נזכיר שבעת שאנחנו כותבים ערך מסוים אנחנו לא יודעים בדיוק (אם לא ציינו) מתי להפסיק לקרוא אותו ולכן צריך בנוסף להגדיר כמה בתים למשל האורך של הרצף צורך בכדי שנקרא את כולם, כנ"ל עבור ערך ההתחלה (Offset).

ישנו המקרה הנוסף שאם ניקח מקרה קיצוני בכדי להמחיש, הוא המקרה שאין בכלל חזרתיות, בסופו של דבר ייכתב 0, עבור אורך וגם עבור ה -Offset ככה שייתכן מצב שאנחנו מרחיבים את גודל הקובץ בנוסף לתיו.

ומכאן באה ההצדקה לאלגוריתם LZSS:

ראשית אנחנו נוותר על רישום אפסים עבור תווים יחידים, אלא נרשום את התו עצמו, שנית, רק ברגע שישנו רצף אנחנו נרשום מאיפה מתחיל הרצף במילון, ומהו אורכו, ללא רישום תו כלשהו מהמילון.
לפני רישום של **תו יחיד נרשום 0**, ולפני רישום **רצף נרשום 1**, סימונים אלה חיוניים עבור תהליך הפיענוח.
בנוסף, נגדיר אורך מינימלי שבו נרצה לרשום רצף, כלומר רצפים קטנים רק מעמיסים על הקובץ הדחוס ונעדיף לרשום אותם בתור תווים בודדים.

הסבר לגבי המימוש:

נתחיל דווקא מהסוף, אל קובץ הפלט אנחנו כותבים אובייקט BitSet, לכתיבה זו יש יתרון הגובר על החיסרון עבור קבצים גדולים.

נתחיל **מהחיסרון**, החלוקה בעת השמירה בזיכרון וגם בעת הכתיבה לקובץ של BitSet דורשת 64 ביטים (Long) בעת הכתיבה של כל הביטים שצברנו במשתנה מסוג זה תהיה השלמה מימין של ביטים דבר שיגדיל את הקובץ.
האמנם במקרה הגרוע ביותר יצטרפו למשתנה האוגר את רצף הביטים (נקרא בקוד StringBuilder encoded) 63 ביטים בכדי להשלים לחלוקה של 64, שהם 8 בתים (ללא ה-metaData שנכתב באמצעות ObjectOutputStream),

אך **היתרון** הוא שישנה חלוקה נוחה שתיראה כך:
במקום שסימון 0 לפני כתיבת בית של תו יחיד תיכתב לקובץ בתור בית, ה0 ייכתב בתור ביט יחיד, (גם עבור 1 וכתיבה של רצף), בתהליך הפענוח יש לנו את היכולת לעבור ביט אחרי ביט, וכך זה יראה:
אנחנו מחזיקים משתנה שיצביע לנו על איזה ביט אנחנו נמצאים, אם נזהה ביט 0, אנחנו קוראים 8 ביטים אחרי (קוראים תיו), ואם **נזהה 1**, תהליך זה מורכב יותר ויראה כך:

קראנו ביט יחיד 1, כעת נקרא כמה בתים צורך ה-0ffset, לשם כך בתהליך הכתיבה הקצנו 2 ביטים (עד יתרון לBitSet שנתן לנו את האפשרות לכתוב רק 2 ביטים) הביטים האלה יאמרו בטווח של 1 עד 3 בתים כמה ה-offset צרך, כלומר אנחנו יכולים מרחק גדול יחסית של מקסימום 8,388,607 בתים אחורה (מסומנים), שהם 50.33 MB, כנ"ל עבור אורך הרצף.

נחזור עכשיו להתחלה (שלב הדחיסה), אנחנו נתחזק חלון חיפוש (היסטוריה/מילון) שבהתחלה צובר MinMatch תווים, שכן עד לרגע זה אנחנו לא יכולים לקרוא רצף, וחלון שיחזיק מספר מוגדר (maxMatch) של בתים שבאמצעותו "נסתכל" קדימה ונזהה רצפים, בכל זיהוי רצף, או תו יחיד חלונות אלה יתקדמו עד לסוף הקובץ (updateBuffers).
נציין בנוסף שבעת הכתיבה (בפיענוח) והקריאה (בדחיסה) השתמשנו ב-BufferdOutPutStream שכן הוא ידוע במהירותו שכן המערכת יודעת מתי לצבור ולשחרר את הבתים לקובץ.

בתחילת הקובץ המקודד נאחסן את ה-metaData של הקידוד, בעת הפיענוח נרצה לדעת מהו גודל חלון החיפוש המקסימלי שכן אנחנו נתחזק אותו במהלך הפיענוח. בנוסף בחרנו לרשום את גודל הביטים שהBitSet השלים לחלוקה ב64, מכיוון שהם לרוב הקבצים יבזבזו פחות מקום מאשר כתיבת גודל הקידוד.

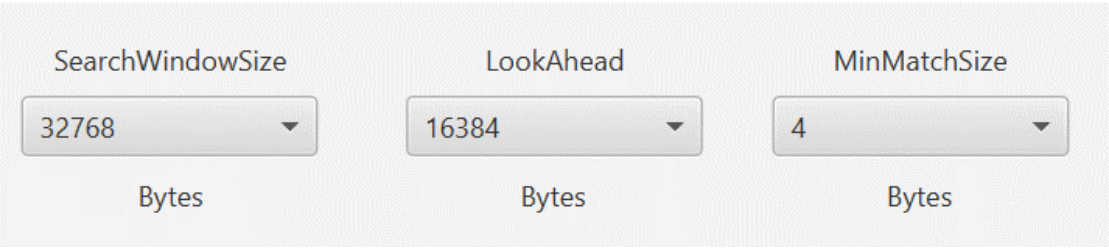
נציין שעבור הממשק יכלנו להשים ערכים גדולים יותר עבור בחירת גדלי החלונות, אך לשם הפשטות, וגם מהעובדה שניתן לשנות פשוט את המערכים ולהוסיף ידנית ערכים במחלקה Controller במערך SearchWindowSizeVal.

השוואה ל-LZ77:

מהאתר: https://www.gutenberg.org/ והקבצים שסופקו לעבודות קודמות

שם הקובץ	גודל מקורי	LZ77	LZSS
American politics.txt	KB4687	KB4041	KB3378
Hector France.txt	KB497	KB483	KB334
OnTheOrigin.txt	KB1027	KB940	KB521
Smiley.BMP	KB184	KB7	KB3
Romeo_and_Juliet_Entire_Play.txt	KB159	KB143	KB96

אלה הן ההגדרות ששמנו:



נסכם בזאת שניתן לראות ולהשתכנע כי האלגוריתם LZSS הוא אלגוריתם יעיל יותר בשל האופטימיזציה על LZ77 והתוצאות לא מפתיעות כי בסך הכל הרעיון הגיוני.

ההשוואה לקוד:
https://github.com/rudbaby/LZ77/blob/master/LZ77/src/bhanu/compression/lz77/LZ77.java