21/12

נושא 6 - דחיסות מילוניות

א) הקדמה

עד כה למדנו שתי גישות כיצד לקודד קובץ: בגישה הסטטית מנתחים את הקובץ לפני הקידוד כדי לקבל סטטיסטיקות ואז מקודדים אותו, ובגישה האדפטיבית (דינאמית) מקודדים כבר במעבר הראשון על הקובץ כל תו לפי מה שראינו עד אליו.

גישה חדשה היא "גישת ההחלפה", שבה עוברים על הטקסט משמאל לימין ומחליפים בלוקים של תווים באצבע (רפרנס) למיקום הקודם בו ראינו בלוק כזה.

ב) דחיסה מילונית

זוהי דחיסה שבה נבנה מילון המכיל בכל כניסה תו בודד או קבוצת תווים וקידוד של התו הבודד או קבוצת התווים. תפקיד המקודד הוא לחלק את הטקסט בצורה כזו שכל חלק נמצא במילון. יכולים להיות המון אפשרויות לחלוקה זו.

הגדרת חלוקה אופטימלית עבור מילון נתון:

בהינתן מילון D, פונקציית קידוד ה (מקבלת מילה w ומחזירה את הקידוד שלה לפי המילון D) וטקסט T, נרצה למצוא חלוקה של T למילים, $T=w_1w_2\dots w_k$, כך שכל מילה $w_i\in D$ וגם סכום אורך כל הקידודים לפי חלוקה זו, כלומר, $\sum_{i=1}^{k}|\text{ה}(w_i)|$ מינימלי.

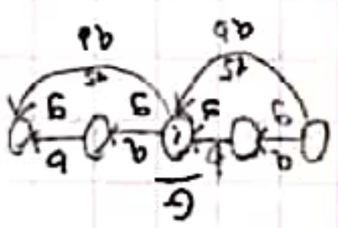
חלוקה אופטימלית: שיטת החלוקה האופטימלית היא לעשות רדוקציה לבעיה מתורת הגרפים. נבנה גרף G שבו הצמתים הם כמספר התווים ב-T ועוד אחד. הצמתים ממוספרים לפי ההופעה שלהם ב-T. כדי להגדיר את הקשתות נעבור על כל זוג צמתים (i,j) ונבדוק אם יש מילה במילון שמתחילה ב-x_i ונגמרת ב-x_{j-1}, אם כן $(i,j)\in E$. הקשת $\overset{i\to j}{\text{מכוונת}}$. משקל כל קשת יהיה אורך הקידוד במילון של המילה. כדי למצוא חלוקה אופטימלית נפעיל על G אלגוריתם דייקסטרה מ-1 עד n+1. כל קשת במסלול שיהיה היא חלק אחד בחלוקה האופטימלית.

$$T = x_1x_2\dots x_n$$
$$G=(V,E)$$
$$V=\{1,2,\dots,n,n+1\}$$
$$(i,j)\in E \iff x_ix_{i+1}\dots x_{j-1}\in D$$
$$w(i,j)=|\text{ה}(x_ix_{i+1}\dots x_{j-1})|$$
$$\text{Partition}=\text{dijkstra}(G,1,n+1)$$

דוגמה:

$T=abab$

$D=\{\underset{9}{a},\underset{9}{b},\underset{15}{ab}\}$



⊛ יש עוד סוגים של חלוקות אך אין אופטימליות (שקופית 3 במצגת).

⊛ בפרק זה נלמד עוד סוגים של דחיסות מילוניות שהן אדפטיביות, כלומר מקודדים אותן במעבר אחד משמאל לימין. ולכן אולי פחות טובות מחלוקה אופטימלית, אך, יש להם יתרון שקל יותר לקודד אותן <u>ואינן דורשות מילון ידוע מראש!</u>

28/12 בס"ד

(28)

ג) LZ77

זהו אלגוריתם קידוד שפותח ע"י שני חוקרים מהטכניון (ziv, Lampel) בשנת 77, ועליו מבוססים אלגוריתמי דחיסה שמשתמשים בהם עד היום בקבצים כמו zip ו-rar. מבוסס על "גישת ההחלפה". באלגוריתם זה מחלקים את הטקסט לשני חלקים: Window ו-look Ahead Buffer. Window זה החלק שכבר קודדנו וה-buffer זה מה שנותר לקידוד. בתחילת האלגוריתם ה-buffer הוא כל הטקסט וה-Window ריק. *עוברים על הטקסט משמאל לימין. בכל פעם שמגיעים לתו בודקים מהי קבוצת התווים שמתחילה ממנו ורציפה שכבר נמצאת ב-Window, ומחליפים קבוצת תווים זו ברפרנס למיקום ב-Window שבו כבר נמצאת קבוצת תווים זו. ה-Window משמש "כמילון" באלגוריתם זה. הרפרנס הוא הקידוד.

בדרך כלל ל-Window ול-buffer יש גודל קבוע. שהוא חזקה של 2. לדוגמה Window בגודל 4096 ו-buffer בגודל 16. עושים זאת כדי שהחיפוש לא יצרוך הרבה זמן.

כיצד נראה רפרנס כזה? הוא מחולק לשלושה חלקים (offset, length, symbol):

- offset - כמה תווים לזוז שמאלה ממיקום נוכחי.
- length - כמה תווים לקרוא מה-offset. אם ה-length גדול מה-offset חוזרים לתחילת ה-offset וממשיכים להעת'
- symbol - התו הראשון שנמצא ב-buffer אחרי קבוצת התווים שעתה העתקנו.

זה נקרא "הדבקה עצמית".

$(0,0,a)(1,98,a)$ זה בעצם a^{100}.

הערה:
במידה ונתקלים בתו c בפעם הראשונה הרפרנס שלו הוא (0,0,c). המשמעות פשוט צריך לעקוב אחרי "הורא'

אלגוריתמי קידוד ופענוח:

```
LZ77 Encoding {
    p = 1                                   // p מייצג מיקום תו ראשון ב-buffer
    while p < s.length // s זהו מערך המכיל את הטקסט
        search for the logest match for
        S[p...] in S[p-W...p-1].  // נניח שנמצא מחרוזת באורך l במרחק m מ-p. W אורך החלון.
        output(p-m, l, S[p+l])
    }   p = p+l+1
}
```

```
LZ77 Decoding {
    p = 1
    for each triple (f, l, c)
        S[p... p+l-1] = S[p-f... p-f+l-1]
        S[p+l] = c
    }   p = p+l+1
}
```

דוגמה:

T = "A_walrus_in_Spain_is_a_walrus_in_Vain"

Encoded String: (0,0,A), (0,0,_), (0,0,w), (0,0,a), (0,0,l), (0,0,r), (0,0,u), (0,0,s), (7,1,i), (0,0,n), (3,1,S), (0,0,p), (11,1,i), (6,2,i), (12,2,a), (21,11,V), (20,3,.)

חסרונות:

(1) ב-LZ77 לא ידוע כיצד למצוא מחרוזת הכי ארוכה שנמצאת ב-Window. יכול להיות מאוד רע לזמן הדחיסה

(2) במקרה שפוגשים תו בפעם הראשונה נקודד (0,0,c) שזהו גודל החלון בביטים ועוד מספר הביטים המייצגים את מספר התווים שאנו צריכים להעתיק מה-Window ועוד 8 ביטים לתו החדש ב-ASCII. אלו הם המון ביטים לחינם, כבר היה עדיף להעתיק את c לבד.

(3) מדוע למצוא ב-buffer מחרוזת הכי ארוכה ב-Window? אולי כדאי דווקא קצרה יותר כי יאפשר לנו בהמשך להעתיק מחרוזת גדולה יותר.

28/12 ㉙ בס"ד

3) LZSS

כדי לענות על חסרונות (1) ו-(2) שתיארנו באלגוריתם LZ77 פותח אלגוריתם LZSS.

באלגוריתם זה יש שני שיפורים מהותיים:

(1) כאשר צריך להעתיק תו אחד בלבד, כגון שנפגשנו בו לראשונה או שהתו הבא אינו מופיע ביחד עם c ב-window, במקום לציין רפרנס צריך נרשום את התו ב-ASCII בלבד. כיצד נבדיל האם זה תו בודד או רפרנס? נוסיף לפני כל קריאה ביט זיהוי, 0 עבור תו אחד ו-1 עבור רפרנס. אם קרא 0 אזי יודע לפרש את 8 הביטים הבאים כתו ASCII ואם קרא 1 קורא מספר ביטים כגודל ה-window הבינארי בשביל ה-offset ועוד גודל ה-buffer הבינארי עבור ה-length. כאשר ה-window וה-buffer מאוד גדולים יכול להיות שגם במקרה של שני תווים או יותר הוא יעדיף להציג אותם ב-ASCII אחד אחרי השני ולא להעתיק אותם. לדוגמה, אם החלון בגודל 2^{14} וה-buffer בגודל 2^4, אזי נדרש $1+14+4=19$ ביטים לייצג רפרנס ואילו שני תווים ב-ASCII דרוש $(1+8)\cdot 2=18$ ביטים. לכן עדיף לא להעתיק אותם כרפרנס

במידה ובוחר להעתיק רק 2 תווים ומעלה, אפשר להחליט שה-length יתחיל מ-2 וכך יהיה אפשר להעתיק יותר.

(2) אלגוריתם זה מתאר כיצד לבצע את החיפוש של מחרוזת הכי גדולה ב-buffer שנמצא ב-window. עבור כל תו שנכנס ל-window לוקחים ממנו ימינה תווים כגודל ה-buffer ומכניסים את המחרוזת שהתקבלה לעץ חיפוש בינארי מאוזן (AVL או אדום שחור). בכל פעם שקוראים תו ב-buffer לוקחים ממנו י מחרוזת בגודל ה-buffer ומחפשים בעץ שנבנה עכשיו את הצומת שיש לה prefix משותף הכי גדול עם מחרוזת נוכחית. כל צומת בעץ צריכה גם את המיקום שלה ב-window כדי שנוכל לבנות את הרפרנס במהירות.* בפועל, משתמשים ב-Hashtable אבל הרעיון הוא דומה.

דוגמה:

A_walrus_in_spain_is_a_walrus_in_vain

A_walrus_in_spa(6,3)is_a(21,11)v(20,3)

דוגמה: עבור buffer בגודל 4 ומחרוזת לקידוד "ababbabaabaa". אם קוראים את התו ה-5 המחרוזת לחיפוש היא "babb", והעץ עד כה הוא:

(*) ניתן לראות שצומת 2 עם פרפיקס הכי גדול באורך 3 במקום 2 בחלון. לכן הרפרנס הבא הוא (3,3)

ab(2,2)(3,3)...

4) LZS

זהו אלגוריתם שפותח בשנת 1994, ועדיין שיפור נוסף ל-LZ77 מעבר לאלו שב-LZSS. לפי אלגוריתם דחיסה זה יש להבדיל בין שני סוגי רפרנסים, כאלו שנמצאים קרוב למיקום הנוכחי בטקסט שאנו מקודדים וכאלו שנמצאים רחוק ממנו.* שני גרסאות לאלגוריתם פותחו ע"י שתי חברות שונות:

המטרה היא לצמצם קידוד של רפרנס.

- <u>Stac Electronics</u> - הרפרנס הקצר מכיל offset בגודל 7 ביטים ויכול להצביע עד 128 תווים אחורה ב-window. והרפרנס הארוך ה-offset בגודל 11 ביטים ויכול להצביע עד 2048 תווים אחורה. המזהה של הרפרנס הראשון הוא "11" בתחילתו והמזהה של השני הוא "10". ככה המפענח ידע כמה ביטים לקרוא ב-offset.
- <u>Microsoft</u> - הבדילו בין 3 סוגים של רפרנסים. ובנוסף כל רפרנס יכול להצביע, בחלון [illegible] שהקידוד לו מ"כ וככה ניתן להגדיל עוד את מרחק ההצבעה. הרפרנס הקצר באורך 6 ביטים ב-offset עם מזהה 11 ויכול להצביע עד 64 תווים אחורה. רפרנס ממוצע באורך 8 ביטים ב-offset עם מזהה 101 ויכול להצביע עד 320 תווים אחורה. והרפרנס אחר עם offset בגודל 11 ב-offset עם מזהה 100 ויכול להצביע עד 2368 תווים אחורה.

תו: 0 [8 ביטים]

רפרנס:

11 [6 ביטים]

101 [8 ביטים]

100 [11 ביטים]

28/12 בס״ד

(30)

1) LZ78

זהו אלגוריתם שפותח בשנת 1978 ע״י אותם חוקרים שפיתחו את LZ77. ההבדל המרכזי ביניהם הוא שב-LZ77 ה-window משמש כמילון ואילו ב-LZ78 בונים ממש מילון. המקודד בונה את המילון תוך כדי פעולת הקידוד ללא שום ידע מוקדם על הטקסט. המילון מורכב מאוסף של מחרוזות (נקראים גם ביטויים). $D=Z_0,Z_1,Z_2,\dots$. מטרת המקודד היא לחלק את הטקסט לחלקים $T=T_1T_2\dots T_n$ כך* שכל $T_i\in D$. במקום כל ביטוי T_i פלט המקודד שם רפרנס למילון המורכב מהצמד (index, symbol), כאשר index זהו המצביע של הביטוי שנמצא במילון ואליו אנו רוצים להפנות ו-symbol זהו התו הבא בטקסט לאחר ביטוי זה.

חלוקה זו נקראת ״רצף״ (sequence).

כיצד עובד האלגוריתם: בתחילה המילון מכיל רק $Z_0=\varepsilon$. בכל פעם שקוראים תו חדש x, מחפשים את האינדקס i של הביטוי שנמצא במילון שהוא הרישא הגדולה ביותר של הטקסט שנותר לקידוד המתחיל בתו שקראנו x. (נשים לב שתמיד יש אינדקס כזה, כי במקרה הגרוע נחזיר 0, שהוא האינדקס של ε). ניקח את הביטוי Z_i שמופיע במילון נוסיף לו את התו הבא y שנמצא בטקסט ונכניס את Z_iy בתור רשומה חדשה למילון שתקבל את האינדקס הפנוי הבא במילון. אז, החלק של Z_iy שנמצא בטקסט T נמיר ברפרנס (i, y).

T= badadadabaab

Index	Phrase	encoding	# of bits
0	ε		
1	b	(0,b)	0+2
2	a	(0,a)	1+2
3	d	(0,d)	2+2
4	ad	(2,d)	2+2
5	ada	(4,a)	3+2
6	ba	(1,a)	3+2
7	ab	(2,b)	3+2

Encoding:(0,b)(0,a)(0,d)(2,d)(4,a)(1,a)(2,b). 26 ביטים

פרט חשוב נוסף, הוא שהמפענח בונה את המילון תוך כדי הפענוח ולכן אין צורך לשלוח את המילון ב-header. תכונה זו מאפשרת למקודד לא לקודד את האינדקס בגודל קבוע אלא משתנה בהתאם לגודל המילון. אנו מגדילים את המילון כל פעם בחזקה של 2*. כל עוד המילון בגודל 2^K נדרש K ביטים לציין את ערך האינדקס. ברגע שהמילון יגדל פי 2 ל-2^{K+1} נדרש K+1 ביטים לייצג את האינדקס.

בהתחלה גודל המילון הוא 1. לכן אין צורך לציין אינדקס.

בדוגמה משמאל, יש רק שלושה תווים (a,b,d) ולכן נדרש שני ביטים לייצג כל תו. ניתן לראות איך מספר הביטים המייצגים את האינדקס גדל ככל שהמילון גדל.

ב) LZW

זהו אלגוריתם דחיסה שמשפר את LZ78 ומשנה אותו מעט. התוספות והשינויים העיקריים הם:

(1) המילון בהתחלה אינו מאותחל עם ε בלבד אלא עם כל התווים הבודדים בקובץ. גודל המילון יהיה השלמה לחזקה של 2 הכי קרובה למספר התווים. בהרבה גרסאות של LZW המילון בהתחלה מאותחל עם כל 256 התווים בטבלת ASCII והגודל הטבלה הוא 512. לכן 9 ביטים לייצג אינדקס

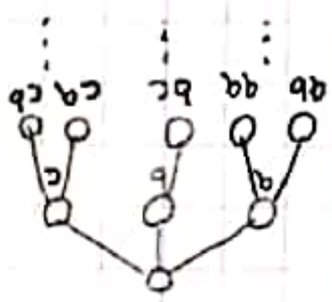
(2) כל רפרנס הוא אינדקס בודד ואינו מכיל את התו הבא. בכל פעם שנמצא מחרוזת מהטקסט שנמצא במילון, נוסיף לה את התו הבא בטקסט p ונכניס למילון כערך חדש. לאחר מכן הקריאה תתחיל מ-p ולא מהת[illegible] כמו ב-LZ78. נשים לב שכדי שמחרוזת תהיה במילון כל הרישות שלה צריכים להיות במילון

בשיטה זו יש צירופים במילון שלעולם לא נפנה אליהם, וכן יהיו צירופים בטקסט שקראנו שלא יכנסו למילון.

4/1 בס״ד

(31)

(3) נתאר כיצד לבצע את החיפוש במילון בצורה יעילה בלי לעבור על כל הערכים. יוצרים עץ לשורש שהוא מייצג את המילון. בתחילה כל התווים הם בנים ישירים של השורש. בכל פעם שמוסיפים מחרוזת מוסיפים אותה לענף שמכיל את כל הפריפיקסים שלה. כמו שתואר באלגוריתם ההמשך כל פעם שנקרא תו נבדוק אם מחרוזת נוכחית נמצאת במילון, אם כן נוסיף עוד תו מהטקסט ונחפש שוב.. כאשר נגיע למחרוזת הכי גדולה נדפיס את האינדקס שלה, נוסיף לה את התו הבא ונכניס כערך חדש למילון. באלגוריתם זה לכל צומת המייצגת מחרוזת יוצאת צומת נוספת המייצגת את סוף המחרוזת (EOS) ומכילה את האינדקס של



```
LZW Encoding {
    D = all single charecters    // D הוא המילון
    W = first char of input
    repeat {
        K = next charecter   // w מייצג מחרוזת נוכחית שמחפשים במילון ו-k זהו התו הבא לאחר w באיטרציה ראשונה k הוא התו השני
        if (EoF)
            output W.index      // רק בסיום הקובץ
        else if (W·K ∈ D)       // משרשרים את k ל-w וקוראים תו הבא.
            W = W·K
        else {
            output W.index      // w לא נמצא במילון. מדפיסים אינדקס w ומכניסים למילון עם התו הבא.
            D.insert(W·K)
            W = K
}   }   }
```

```
LZW Decoder {
    D = all single charecters
    OLD = first input code  // OLD ו-NEW הם אינדקסים במילון
    output OLD.translation
    while not end of input {
        NEW = next input code
        IF NEW ∉ D {        // כאשר NEW הוא אינדקס שלא במילון נצטרך לבנות אותו, להדפיס ולהכניס למילון. S זהו תרגום של NEW.
            S = OLD.translation
            S = S·C
        }
        else
            S = NEW.translation
        output S
        C = first character of S
        D.insert(OLD.translation·C)
        OLD = NEW
    }
}
```

דוגמה לקידוד:

T = "wabba_wabba_wabba_wabba_woo_woo"

Index	Phrase
0	_
1	a
2	b
3	o
4	w
5	wa
6	ab
7	bb

initial dictionary

Index	Phrase
8	ba
9	a_
10	_w
11	wab
12	bba
13	a_w
14	wabb
15	⋮

Encoding: 4 1 2 2 10 5 7 9 11...

$\underset{4}{100}$ $\underset{1}{001}$ $\underset{2}{010}$ $\underset{2}{010}$ $\underset{1}{0001}$ $\underset{0}{0000}$ $\underset{5}{0101}$...

ניתן לראות שכאשר הטקסט גדל ל-16 מקומות הייצוג של כל אינדקס גדל ל-4.

דוגמה לפענוח: Code: 0,1,3,2,4,7,0,9,10,0

OLD	NEW	C	S
0	1	b	b
1	3	a	ab
3	2	c	c
2	4	b	ba
4	7	b	bab
7	0	a	a
0	9	a	aa
10			

dictionary

code	Phrase
0	a
1	b
2	c
3	ab
4	ba
5	abc
6	cb
7	bab
8	baba

initial dictionary

output: a b ab c ba bab a aa ...

נניח כי לא השורה הקודמת והא״ב שלה הוא {a,b,c}.

ניתן לראות שהמפענח בונה את המילון לבד ואינו צריך לקבל כלום ב-header עצמו הזה (במקרה שהם לא משהו מוסכם מראש).

ח) LZ78/LZW vs LZ77/LZSS

בשאלה התחתונה LZW הוא יעיל יותר מ-LZSS בדחיסה של תמונות, בעיקר כאלו שמיוצרים ע״י מחשב שיש בהם חזרות רבות לעומת תמונות מצולמות שבהן אין חזרות. לדוגמה GIF מבוסס על LZW ואכן משמש למטרה זו. דחיסת תמונות שמבוססת על LZSS הוא פ אולם LZSS טוב יותר בדחיסה של קבצי טקסט. יש לכך שתי סיבות והן שב-LZW יש ביטויים במילון שלעולם לא משתמשים בהם, וגם אינו מכיל את כל הביטויים מהטקסט שקראנו. לעומת זאת, LZSS משתמש במילון בכל הטקסט שקראנו וכן הסיכוי למצוא בו רפרנס גבוה יותר.