

תרגיל 9 - קידוד האפמן


תאריך הגשה: יום שלישי, 07.01.2013, 20:55

ראינו במהלך הקורס את אלגוריתם הדחיסה של האפמן (huffman). בתרגיל זה אנו נממש אלגוריתם זה, כך שתוכלו לדחוס ולשחזר קבצים ע"י תוכנה מעשה ידכם.

לשם פשטות נניח שהמידע שאנו רוצים לדחוס הוא רצף של מספרים שלמים בטווח 0 עד 255. זו הנחה שמתקיימת עבור מקרים רבים: לדוגמה עבור ייצוגים מסויימים של תמונות ועבור טקסט שמקודד באמצעות טבלת ASCII. ניתן בקלות להרחיב את התוכנית כך שתוכל לדחוס סוגי מידע נוספים, אך אנו לא נבצע זאת בתרגיל זה. הנחה זו של אלפבית קבוע מאפשרת לנו להשתמש בטבלת קידוד קאנונית. טבלת קידוד קאנונית תופסת הרבה פחות מקום בזכרון מאשר טבלת קידוד מלאה כמו שראינו בכיתה. ניתן לקרוא על טבלת קידוד קאנונית בויקיפדיה או במאמר (בעברית) של yossieb.

בתרגיל זה עליכם לכתוב תוכנית שמממשת קידוד ושחזור נתונים, תוך שימוש בקידוד האפמן. לתרגיל מצורפת תבנית hzlib.py עם ההכרזות על הפונקציות שעליכם לממש. על המימוש לכלול docstring לכל פונקציה.

הפונקציות ב-hzlib.py הן:

- הפונקציה `symbol_count`: קלט: `data`. מחזירה מילון (dictionary) של השכיחויות של התווים השונים ב-`data`. אפשר להסתמך על כך ש-`data` הוא iterable. כמו כן, רצוי להשתמש ב-`collections.Counter`.
- הפונקציה `make_huffman_tree`: קלט `counter`. מחזירה עץ המייצג את קידוד האפמן עבור שכיחויות התווים במילון `counter` (פלט של הפונקציה הקודמת). העץ צריך להיות מיוצג בתור רשומה (tuple) של רשומות באופן רקורסיבי. עלה (= תו) הוא איבר בודד. עץ עם לפחות שני צמתים הוא רשומה שבו האיברים מייצגים את תתי העצים שיוצאים מהשורש. כל תת-עץ כזה יכול להיות איבר בודד אם הוא עלה או רשומה אם זה תת-עץ בגודל 2 או יותר. לדוגמה, העץ  מיוצג על ידי `((a,(b,c)))` בבניית העץ, כאשר יוצרים צומת חדש, הילד השמאלי הוא זה עם המונה הגדול יותר. אם שני המונים שווים, הילד השמאלי הוא זה שהשורש שלו נוצר מאוחר יותר. אם שני הילדים הם עלים (כלומר, תווים הרשומים ב-`counter`), הילד השמאלי הוא המוקדם יותר בסדר הטבעי של העלים ($a < b$ אז `a` קודם ל-`b`). לדוגמה, עבור הקלט `{0:1, 1:1, 2:1, 3:1}` (ארבעת התווים 0,1,2,3, כל אחד הופיע פעם אחת) הפלט צריך להיות `((0,1),(2,3))`. עבור הקלט `{0:1, 1:2, 2:1}`, הפלט צריך להיות `((0,2),1)`.
- הפונקציה `build_codebook`: קלט: `tree`. בונה טבלת קידוד כמילון (dict) שממפה כל תו לרשומה שמחזיק שני ערכי int. הרשומה מהצורה `(length, code)` מייצג את הקידוד של התו באופן הבא. הערך `length` הוא מספר הביטים המשמשים לייצוג הקידוד של התו. הערך `code` הוא מספר טבעי שהייצוג שלו בביטים (באורך `length`) הוא הקידוד של התו. למשל, הקידוד 0101 מיוצג על ידי הזוג (4,5) כי בקידוד ארבעה ביטים ו-0101 בהצגה בינארית הוא 5 בהצגה עשרונית. הקלט לפונקציה הוא עץ קידוד בפורמט של הפלט של הפונקציה הקודמת.
- הפונקציה `build_canonical_codebook`: קלט: `codebook`. מחזירה טבלת קידוד קאנונית בהינתן טבלת קידוד `codebook` (הפלט של הפונקציה הקודמת). טבלת קידוד קאנונית מכילה קידוד של 256 תווים לפי סדר קבוע. (הקידוד עצמו תלוי בתדירויות של התווים הללו). תיאור קידוד קאנוני: עץ קידוד קאנוני הוא עץ שבו העלים העמוקים מופיעים לאחר העלים שאינם עמוקים, והסדר בין העלים בעומק שווה הוא לפי סדר האלפבית. בהנתן עץ קידוד, לכל ערך, נשמור רק את אורך הקידוד. ואז נמין את הערכים לפי אורכי הקידוד בסדר עולה, ושני ערכים בעלי אורך זהה ימויינו לפי סדר האלפבית. לאחר

מכן, כל קוד מוחלף בקוד אחר לפי הכללים הבאים

1. התו הראשון מקבל קידוד בעל אותו אורך שכולו אפסים.
 2. כל תו מקבל את הקוד של התו הקודם בסדרה בתוספת אחד (הוספת ערך, לא שרשור ביט).
 3. אם אורך קידוד התו ארוך יותר מאורך קידוד התו הקודם, אחרי הוספת האחד, יש לשרשר אפסים מימין עד אורך הקידוד הרצוי.
- הפונקציה `build_decodebook`: קלט: `codebook`. מחזירה טבלת שחזור (כ-dictionary) שממפה רצפי ביטים לתווים המתאימים, בהינתן טבלת קידוד `codebook` (פלט של `build_codebook`).
 - הפונקציה `compress`: קלט: `corpus`, `codebook`. מחזירה איטרטור שעובר על הביטים של הקידוד של `corpus` על ידי הטבלה `codebook`. הפלט של האיטרטור הוא 0 או 1 כערכי `int`. הקלט `corpus` נתון כרצף תווים (ע"י אובייקט שהוא `iterable`). הקלט `codebook` הוא טבלת קידוד (פלט של `build_codebook`).
 - הפונקציה `decompress`: קלט: `bits`, `decodebook`. מחזירה איטרטור שעובר על שחזור של רצף התווים המקורי מתוך רצף ביטים מקודד `bits`, בעזרת טבלת השחזור `decodebook`. הקלט `bits` נתון כרצף הביטים המקודדים (ע"י אובייקט שהוא `iterable`). כל ביט הוא ערך `int` של 0 או 1.
 - הפונקציה `pad`: קלט: `bits`. מחזירה איטרטור שעובר על שמיניות הביטים מהסדרה `bits` בתוספת 1 כביט אחרון ואחריו 0-ים במספר הנדרש כדי שסך כל הביטים יתחלק ב-8. הקלט `bits` נתון כרצף הביטים ללא התוספות (ע"י אובייקט שהוא `iterable`). הפלט של האיטרטור שמחזירה `pads` הוא ערכי `byte` (מספרים שלמים בתחום 0 עד 255).
 - הפונקציה `unpad`: קלט: `byteseq`. מחזירה איטרטור שהופך את הפעולה של `pads`. כלומר, האיטרטור עובר על הביטים שמיוצגים ב-`byteseq` לפי סדר ומוריד מהם את התוספת של 1 ואחריו 0-ים בסוף. הקלט `byteseq` נתון כרצף ערכי `byte` (ע"י אובייקט שהוא `iterable`). הפלט של האיטרטור שמחזירה הפונקציה הוא ערכים 0 או 1 ב-`int`.
 - הפונקציה `join`: קלט: `codebook`, `data`. מחזירה איטרטור שעובר כרצף ערכי `byte` על השרשור של טבלת קידוד קאנונית עם רצף מקודד של ביטים. הקלט `codebook` הוא טבלת קידוד קאנונית (פלט של `build_canonical_codebook`). הקלט `data` נתון כרצף של ערכי `byte` (ע"י אובייקט שהוא `iterable`). הפלט של הפונקציה הוא איטרטור שעובר תחילה על הייצוג של `codebook` כרצף ערכי `byte` ואח"כ על רצף הערכים מ-`data`. הייצוג של `codebook` הוא 8 ביטים לכל תו, סה"כ 256 ערכי `byte`. ב-8 הביטים שמייצגים תו רושמים את אורך הקידוד של התו. שימו לב שבטבלת קידוד קאנונית מופיעים 256 תווים בסדר קבוע. אם הא"ב הוא בגודל 256, אז אורך הקידוד של תו לא יכול להיות יותר מ-255, שזה מספר שניתן לייצוג ב-8 ביטים.
 - הפונקציה `split`: קלט: `byteseq`. הפונקציה מפצלת את הפלט של הפונקציה הקודמת לזוג `(data, codebook)`. היא מחזירה `tuple` של שני ערכים. הערך הראשון הוא טבלת הקידוד הקאנונית שמיוצגת ב-256 הבתים הראשונים של הרצף `byteseq`. (הטבלה המוחזרת היא אובייקט מטיפוס `dictionary` - שימו לב שעליכם לשחזר את הקידוד של כל תו מתוך המידע ב-`byteseq` שכולל רק את אורך הקידוד לכל תו.) הערך השני הוא איטרטור שעובר על שאר `byteseq` כרצף ערכי `byte`. הקלט `byteseq` נתון כרצף ערכי `byte` (ע"י אובייקט שהוא `iterable`).

הבהרה: כאשר אנו אומרים שעל פונקציה מסוימת להחזיר איטרטור, הכוונה היא שהמתודה במקום להחזיר ערך היא מפיקה (yield) ערכים, כמו שכבר ראינו. בצורה זו ניתן לעבור על הערכים באמצעות לולאת `for`.

בעזרת המודול `hzip`, עליכם לכתוב תוכנית עזר לדחיסה ושחזור של קבצים. לתרגיל מצורפות התוכניות `hzip.py`

ו-hunzip.py שעליכם להשלים. אלו שתי תוכניות הניתנות להרצה (בעלות main), אחת בכדי לדחוס והשנייה בכדי לשחזר קבצים. על מנת להדפיס תיאור של מה כל תוכנית אמורה לעשות והאפשרויות שלה, ניתן להריץ אותה כפי שצורפה עם האופציה -h:

```
> python3.3 hzip.py -h
```

בקוד שתכתבו עבור השלמת התוכניות הללו עליכם לפתוח קבצים. יש לפתוח אותם עבור קריאה ו/או כתיבה כנדרש. ראו [כאן](#) פרטים נוספים על פתיחת קבצים.

הגשה והנחיות נוספות

ניתן למצוא את כל הקבצים הנלווים לתרגיל [פה](#).

README

- הסבר את דרך ההרצה של הקוד (usage) והאפשרויות השונות.
- בנוסף להנחיות הרגילות, עליכם לענות על השאלות הבאות:
 - מדוע כיווץ צורך זכרון רב יותר מאשר פתיחה (decompression)?
 - איך ניתן לכווץ ללא שימוש רב בזכרון?

יצירת קובץ Tar

בשלב זה, אמורים להיות בידכם הקבצים הבאים:

- hzlib.py
- hzip.py
- hunzip.py
- README

על קובץ ה-Tar להכיל קבצים אלה בלבד. צרו קובץ Tar בשם ex9.tar מהקבצים האלה ע"י הפקודה:

```
tar cvf ex9.tar hzlib.py hzip.py hunzip.py README
```

הגשת הקובץ

- הגישו את הקובץ בלינק הרלוונטי באתר הקורס.
- לאחר הגשת הקובץ, אמור להגיע אי-מייל שמכיל את תוצאות הבדיקה האוטומטית וקובץ ההגשה שיעבור לבדיקה ע"י הבודק האנושי.
- קראו בעיון את הקובץ שהתקבל, וודאו שכל הקבצים מופיעים בצורה מלאה, ברורה ובגרסתם האחרונה.

הרצת טסטים ידנית

ישנן שתי דרכים להרצת הטסטים ידנית (ללא הגשה למודל):

- שימו את הקובץ ex9.tar בתיקייה ריקה, היכנסו לתיקייה והריצו את הפקודה:
~intro2cs/bin/testers/ex9 ex9.tar
- הורידו את הקובץ [ex9testing.tar.bz2](#). פתחו את הכיווץ (tar -xjv ex9testing.tar.bz2) של הקובץ ועקבו אחר ההוראות בקובץ TESTING.