Toteutusdokumentti

1 Ohjelman yleisrakenne

Ohjelman rakenne on hyvin yksinkertainen. Pakkauksia on neljä:

1. `main` sisältää pääohjelman, joka lukee asetukset ja käynnistää pakkauksen/purkamisen. Ainoa luokka on `Main`.
2. `util` sisältää joitain itse toteutettuja yleishyödyllisiä luokkia:
   - `BitInputStream`, `BitOutputStream`: mahdollistavat biteittäin tapahtuvan streamien lukemisen ja kirjoittamisen
   - `List`, `Set`: itse toteutetut `ArrayList` ja `HashSet`
   - `Pair`: kahden objektin pari
   - `Math`: muutama simppeli matematiikkafunktio
   - `Options`, `Option`: komentoriviltä annettavien parametrien lukeminen
3. `huffman` sisältää Huffman-koodausalgoritmit. Luokat:
   a) `Huffman`: varsinaset pakkaus/purkufunktiot sekä tarvittavat apufunktiot. Kaikki staattisia metodeja.
   b) `HuffmanTree`, `HuffmanTreeNode`: Huffman-puu (tavallinen binääripuu paitsi että lehtiin pääsee suoraan käsiksi taulukon kautta).
   c) `HuffmanHeap`: itse toteutettu erikoistapaus `PriorityQueuesta` (minimikeko). Käytetään Huffman-puun konstruoinnissa.
4. `lzw` sisältää LZW-koodausalgoritmit: Luokat:
   a) `LZW`: varsinaiset pakkaus/purkufunktiot sekä tarvittavat apufunktiot. Kaikki staattisia metodeja.
   b) `LZWDictionary`, `LZWDictionaryEntry`: pakkausvaiheessa käytettävä sanasto (prefiksipuu).

2 Saavutetut aika- ja tilavaativuudet

Luokkien List, Set ja HuffmanHeap vaativuudet ovat samat kuin Javan valmiilla ArrayList, HashSet ja PriorityQueue -luokilla.

2.1 Huffman

2.1.1 Pakkaus

Seuraava olettaa, että syöte on luettavissa tavu kerrallaan `read()`-funktiolla ja funktio `write()` kirjoittaa tulosteeseen listan bittejä.

```
compress():
  frequencies = calculateFrequencies()
  tree = buildTree(frequencies)
  while !EOF:
    b = read()
    write(findCode(tree, b))
```

Funktio `calculateFrequencies()` on yksinkertainen:

```
calculateFrequencies():
  result = int[256]
  while !EOF:
    b = read()
    result[b]++
  return result
```

Kyseessä on $O(n)$-aikainen ja $O(1)$-tilainen funktio.

Funktio `buildTree()` muodostaa merkkitaajuuksista Huffman-puun:

```
buildTree(frequencies):
  queue = new MinHeap()
  for b = 0 ... 255:
    queue.push(TreeNode(frequencies[b]))

  while queue.size >= 2:
    a = queue.pop()
    b = queue.pop()
    queue.push(new Node(a.frequency + b.frequency))

  return queue.pop()
```

Ensimmäinen simukka käydään aina 256 kertaa läpi. Toisessa silmukassa jonon alkioiden määrä vähenee aina yhdellä, joten sekin käydään (noin) 256 kertaa läpi. Lopputuloksena `buildTree()` on vakioaikainen ja -tilainen.

Sitten vielä `findCode()`:

```
findCode(node):
  result = new List()
  while node.parent:
    if node == node.parent.right:
      res.add(true)
    else:
      res.add(false)
    node = node.parent
  return res.reverse()
```

Kuten metodista `buildTree()` nähdään, Huffman-puun korkeus on korkeintaan 256. Tästä seuraa, että `findCode()`:n silmukka ajetaan kettä orkeintaan 256 kertaa ja palautettava lista on korkeintaan näin pitkä. Siis kyseessä on vakioaikainen- ja tilainen funktio. Lopputuloksena `compress()` on $O(n)$-aikainen ja $O(1)$-tilainen.

2.1.2 Purku

Seuraava olettaa, että taulukko `freqs` sisältää alkuperäisen (pakkaamattoman) syötteen merkkitaajuudet. Se voidaan tallentaa esimerkiksi pakatun tiedoston alkuun ja lukea siitä vakioajassa. Funktio `read()` lukee syötettä bitti kerrallaan ja `write()` kirjoittaa tulosteeseen annetun tavun.

```
decompress():
  tree = buildTree(freqs)
  node = tree.root
  while !EOF:
    if read() == 0:
      node = node.left
    else:
      node = node.right
    if node.left == null:
      write(node.character)
      node = tree.root
```

Aiemmin jo nähtiin, että `buildTree()` on vakioaikainen ja -tilainen. Loppu purkufunktiosta tekee syötteen joka bitin kohdalla kerran jotain vakioaikaista, joten koko funktio on ajaltaan $O(n)$. Tilavaativuus on $O(1)$.

2.2 LZW

2.2.1 Pakkaus

Perusversio LZW-pakkausalgoritmista on seuraava:

```
compress():
  dict = LZWDictionary()
```

```
currentString = []
while !EOF:
  b = read()
  if not dict.hasCodeFor(currentString + b)
    write(dict.getCodeFor(currentString))
    dict.addCodeFor(currentString + b)
    currentString = [b]
  else:
    currentString += b
if currentString.notEmpty():
  write(dict.getCode(currentString))
```

Algoritmi on työssä toteutettu hieman toisella tavalla, nimittäin siinä lista `currentString` on "sisäänrakennettuna" luokkaan `LZWDictionary`. Tällöin funktioiden `xxxCodeFor()` vaativuudet ovat vakioita; muuten ne riippuisivat listan `currentString` pituudesta. Listan käyttäminen kuitenkin helpottaa funktion ymmärtämistä. Pakkausfunktio käy syötteen jokaisen merkin kerran läpi ja tekee joka merkin kohdalla jotain vakioaikaista, joten funktion aikavaativuus on $O(n)$. Sanastoon `dict` voi tallettaa jonkin kiinteän määrän sanoja, joiden pituus voi vaihdella. Pahimmillaan voi käydä niin, että syötteen lähes jokainen merkki on sanastossa kaksi kertaa. Näin ollen tilavaativuus on myös $O(n)$.

2.2.2 Purku

LZW-purkualgoritmin perusversio:

```
decompress():
  dict = List<List<Integer>>()
  inDict = HashSet<Integer>()
  lastOutput = []
  while !EOF:
    code = read()
    decoded = dict[code]
    toDict = lastOutput + decoded[0]
    if not inDict(toDict):
      dict.set(nextCode++, toDict)
      inDict.add(toDict)
    for i in decoded:
      write(i)http://en.wikipedia.org/wiki/Lossless_compression
```

Pseudokoodista on jätetty pois LZW-purkualgoritmin poikkeustapaus (koodia ei ole sanastossa), mutta se ei muuta vaativuuksia. Koshttp://en.wikipedia.org/wiki/Lossless_compressionka algoritmi toimii oikein (todistus sivuutetaan), niin aika- ja tilavaativuudet ovat $O(n)$, missä n on *alkuperäisen* (pakkaamattoman) syötteen koko.

2.2.3 Viritetyt versiot

LZW-algoritmin perusversioon on työssä tehty kaksi parannusta: muuntuvat koodisanojen koot (koolla on ennalta annettu yläraja) ja täyttyneen sanaston tyhjennys. Kumpikaan muutos ei vaikuta vaativuusluokkiin.

3 Työn mahdolliset puutteet ja parannusehdotukset

Yksi tapa parantaa pakkausalgoritmia voisi olla parantaa tapaa, jolla sanaston täyttyminen käsitellään. Tällä hetkellä täyttynyt sanasto tyhjennetään välittömästi ja aloitetaan alusta. Parempi tapa voisi olla tarkkailla pakkaussuhdetta ja tyhjentää sanasto vasta, kun pakkaussuhde laskee alle jonkin rajan. Toinen tapa olisi, että kun sanasto täyttyy, niin poistetaan siitä kauimmin sitten lisätty (tai luettu) sana. Tällöin ei tosin koodisanan kokoa voi resetoida, vaan se pysyy maksimissa loppuun asti.

4 Lähteet

Wikipedia ja muutama sivu, jotka Google löytää esim. hakusanalla ”lzw” (en muista tarkemmin...).