UNIVERSITEIT
GENT

Brent Van Wynsberge
3e bachelor Informatica, Universiteit Gent
Algoritmen en Datastructuren III
Stamnummer: 01201853

5 december 2017

Huffman compressie

Project Algoritmen en Datastructuren III

Inhoudsopgave

1. Algoritmen

In dit onderdeel bekijken we de pseudocode van alle geïmplementeerde algoritmen. Wanneer er een niet triviale operatie opgeroepen wordt, wordt deze beschreven in het onderdeel 'Algemene Operaties'.

De WRITE(foo) operatie heeft als betekenis: schrijf foo weg naar stdout. De operatie abstraheert het eventuele gebruik van buffers of de alignering van de bits weg.

De READ operatie heeft als betekenis: lees de volgende byte van stdin. Analoog met WRITE abstraheert dit het gebruik van buffers weg en doet dit alsof de tekst altijd byte per byte wordt ingelezen.

1.1 Notaties

- **S**: het aantal symbolen in het alfabet[1] (van de tekst).
- **n**: het aantal bytes in het bestand[2].

[1] opmerking: $\forall S : S \leq 256$, aangezien we telkens 1 byte inlezen
[2] opmerking: $\forall S, n : n \geq S$

1.2 Statische huffman

1.2.1 Algemene operaties

Invoer lezen

Algoritme 1: incrementWeight

Data: input, byte
Resultaat: Gewicht van $byte$ wordt verhoogd en $nodes$ wordt gesorteerd.

```
1  node ←FINDNODE(byte)
2
3  if ¬node then
4  |   node.value ← byte
5  |   node.weight ← 0
6  |   node.next ← input.nodes
7  |   input.nodes ← node
8  end
9
10 node.weight++
11 while node.next ∧ node.weight > node.next.weight do
12 |   SWAP(node,node.next)
13 |   node ← node.next
14 end
15
```

FINDNODE overloopt de gelinkte lijst tot het element gevonden wordt of tot de lijst stopt. Nadien wordt het gewicht verhoogd en wordt de node verder in de lijst verplaatst tot het minstens even groot is als de volgende. In het slechtste geval wordt de gehele lijst 1 keer overlopen. De complexiteit is dus $\mathcal{O}(S)$.

Algoritme 2: readInput

Resultaat: Stdin wordt in input gelezen en de gewichten worden berekend.

```
1  while (byte ←READ( )) ≠ EOF do
2  |   input.content[input.size++] ← byte
3  |   INCREMENTWEIGHT(input, byte)
4  end
5
```

Deze operatie heeft complexiteit $\mathcal{O}(S * n)$.

Boom opbouwen

Algoritme 3: reverseNodes

Data: input
Resultaat: De symbolen met gewichten in omgekeerde (in ons geval: dalende) volgorde.

```
1  head ← input.nodes
2  prev ← ∅
3  while head do
4  |   next ← head.next
5  |   head.next ← prev
6  |
7  |   prev ← head
8  |   head ← next
9  end
10
11 input.nodes ← head
```

Deze operatie heeft complexiteit $\mathcal{O}(S)$.

Algoritme 4: buildStack

Data: input
Resultaat: Een stack met alle bladeren van de huffman-boom wordt opgebouwd.

```
1  REVERSENODES(input)
2  node ← input.nodes
3  stack ← ∅
4
5  while node do
6  |   tree_node ← new tree_node()
7  |   tree_node.left, tree_node.right, tree_node.parent ← NULL
8  |   tree_node.value, tree_node.weight ← node.value, node.weight
9  |
10 |   PUSH(stack,tree_node)
11 |   node ← node.next
12 end
```

Indien er slechts 1 symbool is voegen we nog een lege node toe zodat we altijd een code kunnen schrijven.

De complexiteit van deze operatie is $\mathcal{O}(S)$.

Algoritme 5: buildTree

Data: input
Resultaat: De huffman-boom wordt opgebouwd.

```
1  stack ←BUILDSTACK(input)
2  tmp_stack ← ∅
3  while SIZE(stack)> 1 do
4  |   node1 ← pop(stack)
5  |   node2 ← pop(stack)
6  |
7  |   tree_node ← new tree_node()
8  |   tree_node.left, tree_node.right ← node1, node2
9  |   tree_node.weight ← node1.weight + node2.weight
10 |
11 |   while PEEK(stack).weight < tree_node.weight do
12 |   |   PUSH(tmp_stack, POP(stack))
13 |   end
14 |
15 |   PUSH(stack, tree_node)
16 |   PUSHALL(stack, tmp_stack)              // evenveel iteraties als 11.
17 end
18
19 return POP(stack)
```

De binnenste lus (11) zal een maximum aantal keer overlopen worden, als de som groter is dan alle reeds bestaande toppen. De tweede stack zal dan maximaal met $\lceil \log \mathbf{S} \rceil$ elementen gevuld worden, en die lus zal dan ook zoveel keer runnen.

De buitenste lus maakt alle toppen die geen bladeren zijn aan. Als we dus weten hoeveel toppen (zonder bladeren) er in de huffman-boom zijn dan weten we hoeveel iteraties er plaats vinden.

We weten dat de huffman-boom een perfecte binaire boom is omdat hij anders niet optimaal zou zijn. We weten dat voor een perfecte binaire boom geldt dat $n = 2 * b - 1$ met b het aantal bladeren of dus $b = \mathbf{S}$. Maw. zijn er $n - l = 2 * \mathbf{S} - 1 - \mathbf{S} = \mathbf{S} - 1$ niet-blad toppen in een perfecte binaire boom.

De complexiteit van de BUILDSTACK operatie is dus $\mathcal{O}(S * logS)$.

Woordenboek opbouwen

Algoritme 6: buildDictionary

Data: code [] dictionary, **bit[]** path, node, index
Resultaat: Een dictionary met een referentie naar de prefixcode voor elk symbool wordt opgebouwd.

```
1  if node.left ∧ node.right = ∅ then
2      code ← newCode()
3      code.code ← path
4      code.key ← node.value
5      dictionary[code.key] ← code
6  else
7      path[index] ← 0
8      BUILDDICTIONARY(dictionary, path, node.left, index+1)
9      path[index] ← 1
10     BUILDDICTIONARY(dictionary, path, node.right, index+1)
11 end
```

De gehele boom wordt recursief doorlopen, er zullen dus $\log(2 * l - 1)$ operaties uitgevoerd worden. De complexiteit van deze operatie is dus $\mathcal{O}(\log S)$.

De boomstructuur encoderen

Algoritme 7: printTree

Data: node,**bit[]** tree, **char[]** symbols, i
Resultaat: De huffman-boom wordt in stringformaat omgezet.

```
1  if node.left ∧ node.right = ∅ then
2      symbols[i] ← node.value
3      tree[i] ← 1
4      return i++
5  else
6      symbols[i] ← ∅
7      tree[i] ← 1
8      i ←PRINTTREE(dictionary, path, node.left, index+1)
9      return PRINTTREE(dictionary, path, node.right, index+1)
10 end
```

Analoog met hierboven wordt de boom recursief doorlopen, ook deze operatie heeft dus complexiteit $\mathcal{O}(\log S)$.

1.2.2 Encoderen

Algoritme 8: encodeInput

Data: input, codes
Resultaat: De bytes worden geëncodeerd weggschreven.

```
1 for byte in input do
2 |   WRITE(codes[byte].code)
3 end
```

Aangezien de WRITE(o)peratie in $\mathcal{O}(1)$ tijd verloopt heeft deze operatie complexiteit $\mathcal{O}(n)$.

Algoritme 9: encode

Resultaat: De huffman-boom wordt opgebouwd en de tekst wordt hiermee geëncodeerd.

```
1  input ←READINPUT( )
2
3  tree, max ←BUILDTREE(input)
4  codes ← ∅
5  BUILDDICTIONARY(codes, ∅, tree, 0)
6
7  symbols ← ∅
8  tree_coded ← ∅
9  PRINTTREE(tree, tree_coded, symbols, 0)
10 WRITE(tree_coded)
11 WRITE(symbols)
12
13 ENCODEINPUT(input, codes)
14
15 WRITE(amount_of_last_byte_to_ignore)
```

De duurste bewerking is duidelijk de READINPUT bewerking, met complexiteit $\mathcal{O}(S * n)$. Aangezien we een bovengrens voor **S** gevonden hebben (256) kunnen we dit echter als constante beschouwen.

De ENCODE operatie heeft dus complexiteit $\mathcal{O}(n)$.

1.2.3 Decoderen

1.3 Adaptive huffman

1.3.1 Algemene operaties

1.3.2 Encoderen

1.3.3 Decoderen

1.4 Adaptive huffman met sliding window

1.4.1 Algemene operaties

1.4.2 Encoderen

1.4.3 Decoderen

1.5 Two pass adaptive huffman

1.5.1 Algemene operaties

1.5.2 Encoderen

1.5.3 Decoderen

1.6 Bloksgewijze adaptive huffman

1.6.1 Algemene operaties

1.6.2 Encoderen

1.6.3 Decoderen

2. Datasets

3. Experimenten

4. Besluit

4.1 Statische huffman

4.2 Adaptive huffman

4.3 Adaptive huffman met sliding window

4.4 Two pass adaptive huffman

4.5 Bloksgewijze adaptive huffman