UNIVERSITEIT
GENT

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

7 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 (of bit afhv. de context) 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

Algoritme 10: encode

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

```
1  input ←READINPUT( )     // Zonder characters bij te houden (O(n)).
2  tree_size ←READ(input)
3  chars ← 0
4  tree ← ∅
5  for i in {0,tree_size} do
6  |  tree[i] ←READ(input)
7  |  if tree[i] = 1 then
8  |  |  chars++
9  |  end
10 end
11
12 char_list ← ∅ for i in {0,chars} do
13 |  char_list[i] ←READ(input)
14 end
15 root ←REBUILDTREE(tree, char_list)     // Inverse van printTree O(log S).
16 current ← root
17
18 for bit in READ(input) do
19 |  if current.left ∧ current.right = ∅ then
20 |  |  WRITE(current.value)
21 |  end
22 |  if bit = 1 then
23 |  |  current ← current.left
24 |  else
25 |  |  current ← current.right
26 |  end
27 end
```

De laatste bits moeten nog genegeerd worden, maar aangezien we de hele tekst lezen kunnen we in $\mathcal{O}(1)$ weten hoeveel bits we moeten negeren en laten we dit in het algoritme weg.

De duurste operatie bij het decoderen is het lezen van de overige bits en de corresponderende huffman-boom te overlopen.

Voor elke character in de originele tekst wordt de boom 1 keer overlopen. Decoderen heeft dus een complexiteit van $\mathcal{O}(n * \log S)$ of $\mathcal{O}(n)$ indien we **S** constant nemen.

Wat na deze analyse van statische huffmancodering opvalt is dat alle operaties gebeuren ifv. het alafbet en niet de tekst. En dat we dit alfabet constant kunnen nemen. Dit zal ook zo zijn voor de andere algoritmen dus we zullen altijd een complexiteit van $\mathcal{O}(n)$ bekomen. Daarom worden de algoritmen hieronder kort beschreven, maar de complexiteiten niet berekend. Verder zullen we de runtimes vergelijken om de tijdsverschillen tussen de algoritmen te bekijken.

1.3 Adaptive huffman

Het adaptieve huffman algoritme werkt online. Dwz. dat de tekst character per character vertaald wordt zonder dat de tekst op voorhand doorlopen moet worden. Het algoritme staat al uitgebreid in de cursus beschreven, dus hieronder gaan we er niet te ver op in.

Het voornaamste verschil is dat we een gelinkte lijst van toppen gebruiken ipv. ordernummers.

Een verdere optimalisatie zou kunnen zijn dat we per gewicht een gelinkte lijst van toppen opslaan in een array, maar deze werd niet geïmplementeerd.

1.3.1 Encoderen

Algoritme 11: encode

Resultaat: Het bestand wordt character per character geëncodeerd.

```
1 while (byte =READ( )) ≠ EOF do
2     if ¬ALREADYREAD(byte) then
3         WRITE(nng)
4         WRITE(byte)
5     else
6         WRITECODE(byte)
7         UPDATE(tree, byte)
8     end
9 end
```

1.3.2 Decoderen

Algoritme 12: decode

Resultaat: Het bestand wordt character per character geñcodeerd.

```
1  current ← root
2  while (bit =READ( )) ≠ EOF do
3      if current == nng then
4          byte ←READBYTE( )
5          WRITE(byte)
6          UPDATE(byte, tree)
7      else
8          if current.left, current.right = ∅ then
9              WRITE(current.value)
10             UPDATE(current.value)
11             current = root
12         end
13
14         if bit =′ 1′ then
15             current = current.left
16         else
17             current = current.right
18         end
19     end
20 end
```

1.4 Adaptive huffman met sliding window

Dit algoritme lijkt sterk op het adaptive huffman algoritme. Het grootste verschil met het adaptive algoritme is dat we nu gebruik maken van een window. Dit window implementeren we als een byte buffer waar we altijd op de volgende plaats een byte wegschrijven. Indien er reeds een byte op die plaats in de buffer zat, verminderen we de frequentie van die byte en schrijven de nieuwe byte naar de buffer. Deze stap wordt altijd gedaan nadat we de stappen in het adaptive algoritme toepassen.

We verlagen de frequenties op het pad naar het symbool van de byte top-down.

We beschrijven de REDUCEFREQUENCY operatie hieronder:

Algoritme 13: reduceFrequency

Data: currentNode (heeft referentie naar ouder, kinderen en de volgende in de ordernummerlijst), destinationNode, nng (eerste node in de gelinkte lijst)
Resultaat: De frequentie van alle toppen op het pad naar het blad wordt met 1 verminderd.

```
1  oldParent ← currentNode.parent
2  current ← nng
3  while current.weight < currentNode.weight do
4  |  current ← current.next
5  end
6
7  oldCurrent ← currentNode.tree_reference
8  if current ≠ currentNode ∧ ¬ISPARENT(current, currentNode) then
9  |  SWAP(current, currentNode)
10 end
11
12 if oldCurrent.parent == oldParent then
13 |  oldCurrent−−
14 else
15 |  while NOTACHILD(oldParent, destinationNode) do
16 |  |  oldParent.weight++
17 |  |  oldParent ← oldParent.parent
18 |  end
19 |  oldCurrent ← oldParent
20 end
21
22 if oldCurrent.weight == 0 then
23 |  DELETE(oldCurrent, nng)
24 else if oldCurrent ≠ destinationNode then
25 |  if ¬NOTACHILD(oldCurrent.left, destinationNode) then
26 |  |  REDUCEFREQUENCY(oldCurrent.left, destinationNode, nng)
27 |  else
28 |  |  REDUCEFREQUENCY(oldCurrent.right, destinationNode, nng)
29 |  end
30 end
```

NOTACHILD kijkt of het 2^e argument op het pad ligt van het eerste.

1.5 Two pass adaptive huffman

Hier bouwen we de boom op met de BUILDTREE[alg. 5] operatie. Nadien passen we na het encoderen van elke byte de REDUCEFREQUENCY[alg. 13] operatie toe. Het decoderen verloopt analoog.

1.6 Bloksgewijze adaptive huffman

Dit algoritme verloopt volledig analoog met adaptive huffman, we houden enkel bij hoeveel bytes we al geëncodeerd hebben en beginnen opnieuw wanneer dit de block_size is. Om te decoderen tellen we dan opnieuw de gedecodeerde bytes.

2. Datasets

Om de algoritmen met elkaar te vergelijken maken we gebruik van een aantal soorten datasets: tekst, binair, optimaal.

2.1 Tekst

De tekstbestanden worden gebruikt om te kijken hoe elk algoritme presteert wanneer het een gewone tekst moet encoderen en decoderen. Hiervoor worden enkele boeken van project gutenberg[1] in .txt formaat gebruikt.

De entropie van tekstbestanden is relatief laag (sommige bytes komen veel vaker voor dan andere), dus we vermoeden dat als we codewoorden van 1 byte (1 character) gebruiken we de veel voorkomende symbolen sterk zullen kunnen comprimeren en elk algoritme dus goed zal presteren.

2.2 Binair

Hiervoor gebruiken we enkele afbeeldingsbestanden. Wanneer we binaire bestanden op byte-niveau lezen krijgen we willekeurige contextloze informatie te zien. De entropy van deze data is dus hoog, het zal maw. moeilijker zijn om bepaalde bytes met minder bits te kunnen wegschrijven. We verwachten dat elk algoritme hier relatief slecht presteert.

[1] https://www.gutenberg.org

2.3 Optimaal

Voor onderstaande gespecialiseerde algoritmen bekijken we een dataset waar het in theorie optimaal zou moeten presteren, en vergelijken we dit met de andere algoritmen om te zien of dit effectief het geval is.

Adaptive & statische huffman kunnen we beschouwen als 'niet gespecialiseerde' compressiealgoritmen, in principe zullen ze beter presteren wanneer alle bytes uniform over de tekst verdeeld zijn (maar de frequenties genoeg verschillen). In de praktijk zal dit voor tekstbestanden bijna altijd het geval zijn en is het maken van een optimale dataset niet echt relevant.

Bloksgewijze adaptive huffman heeft sterke gelijkenissen met het 'sliding window' algoritme en zal dus vermoedelijk ook redelijk goed werken voor die optimale dataset. Ook zijn de voordelen van bloksgewijze huffman niet zozeer de compressiefactor en is die iets minder belangrijk.

2.3.1 Sliding window

Dit algoritme zou optimaal moeten presteren wanneer de tekst over het algemeen veel characters heeft die ongeveer even vaak voorkomen, maar waar er lokaal (binnen het window) veel verschillen tussen de frequenties zijn.

sliding.opt is een textbestand waar elke character even vaak voorkomt maar waar ze telkens geclusterd zijn in een reeks die een aantal keer groter is dan de window size.

2.3.2 Two pass

Dit algoritme zou het beste moeten werken wanneer de frequenties ongeveer hetzelfde zijn, maar er bepaalde characters vaker voorkomen in het begin dan op het einde.

two_pass.opt: we herhalen een character 'a' een aantal keer. Nadien herhalen we dat character niet meer maar gebruiken we andere characters die een willekeurig aantal keer voorkomen. We zorgen ervoor dat 'a' significant vaker voorkomt dan de andere characters.

3. Metrieken

Om de algoritmen te vergelijken en te kijken wat in de realiteit hun nut is, gebruiken we 2 metrieken. De tijd en de compressiefactor (ratio).

3.1 Tijd

Deze metriek is vooral belangrijk voor de online algoritmen, omdat het vaak belangrijker is om de data voldoende snel over te brengen dan om de hoogst mogelijke compressie te bereiken (vb.: datastream die over het internet verzonden wordt).

3.2 Compressiefactor

De compressiefactor ($\frac{originele\ bestandsgrootte}{gecodeerde\ bestandsgrootte}$) is een maatstaf voor hoeveel maal kleiner het bestand wordt na coderen. Deze metriek zal belangrijker zijn voor bestanden die gedurende lange tijd opgeslagen zullen worden en niet direct gelezen hoeven te worden (vb.: een back-up). Voor deze bestanden is de decodeertijd ook vaak belangrijker dan de encodeertijd.

4. Experimenten

We vergelijken de verschillende algoritmen met elkaar, we doen dit door de algoritmen op de verschillende datasets toe te passen en de voorafgenoemde metrieken toe te passen.

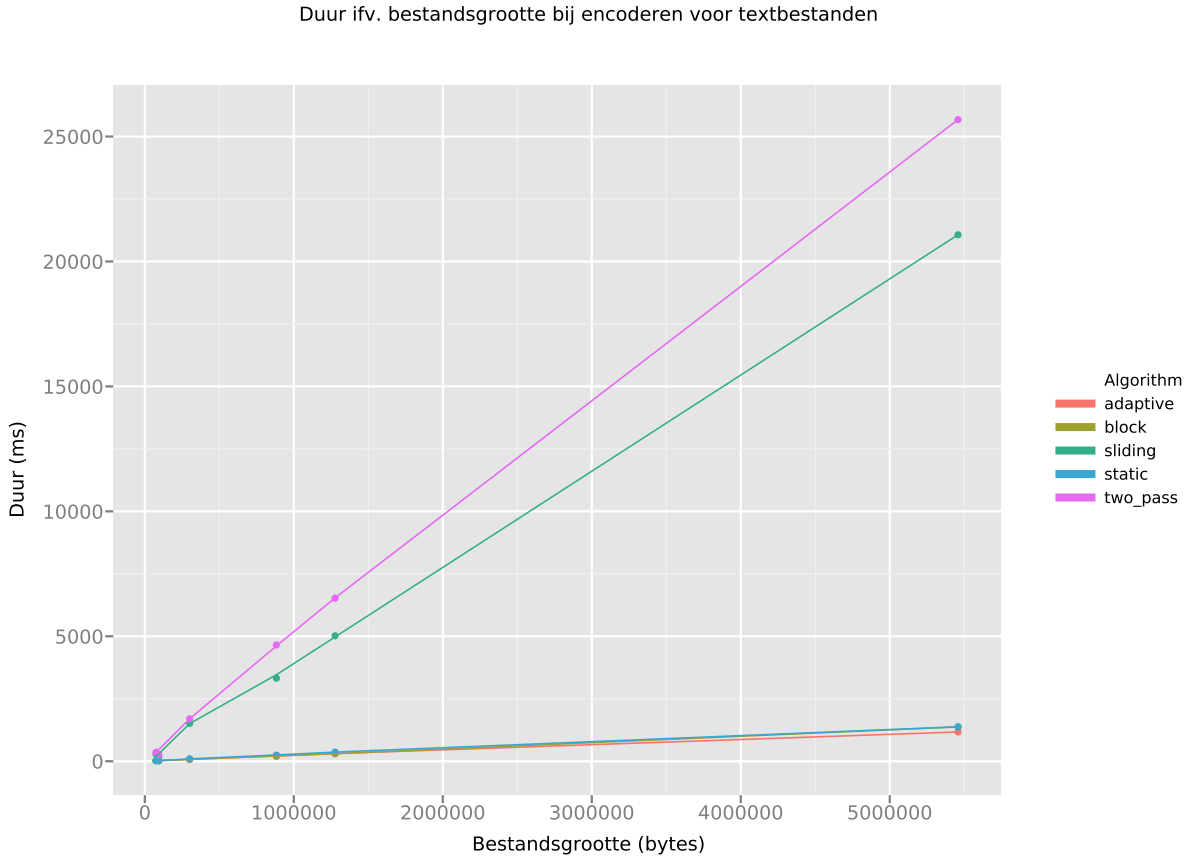
4.1 Duur

We vergelijken hieronder de duur van het encoderen en decoderen van verschillende bestanden met verschillende grootte. We maken hiervoor het onderscheid tussen binaire en tekstbestanden, dit omdat we reeds weten dat binaire bestanden slechter te coderen zijn.

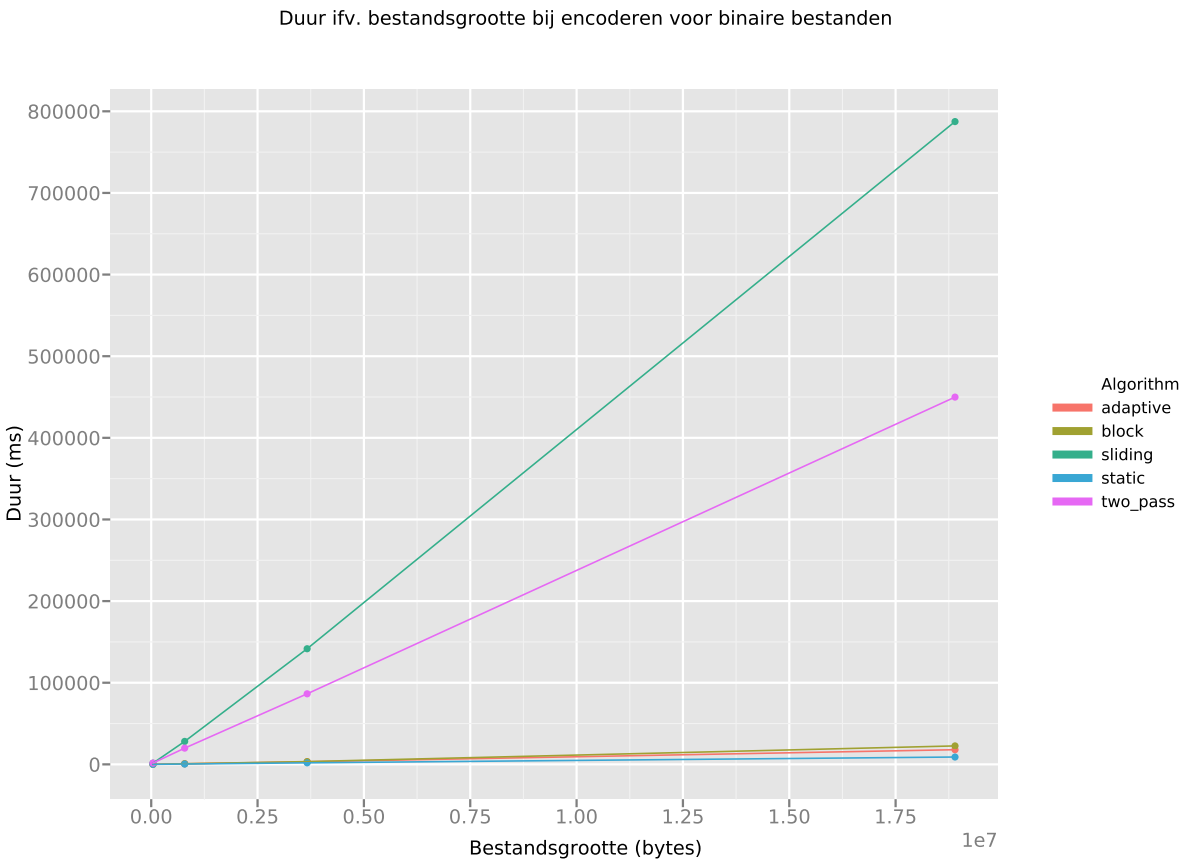
4.1.1 Encoderen

In onderstaande grafieken merken we op dat tekstbestanden sneller encoderen dan de binaire bestanden. Dit valt te verklaren door het feit dat er meerdere verschillende soorten bytes voorkomen dan de 127 ascii characters in de tekstbestanden. De huffman-bomen voor binaire bestanden zullen over het algemeen groter zijn en de operaties op die bomen dus ook duurder.

Wat we ook opmerken is dat het 'sliding window' en het 'two pass' algoritme opvallend veel trager zijn dan de andere algoritmen. Dit komt door het feit dat de REDUCEFREQUENCY een relatief dure operatie is.

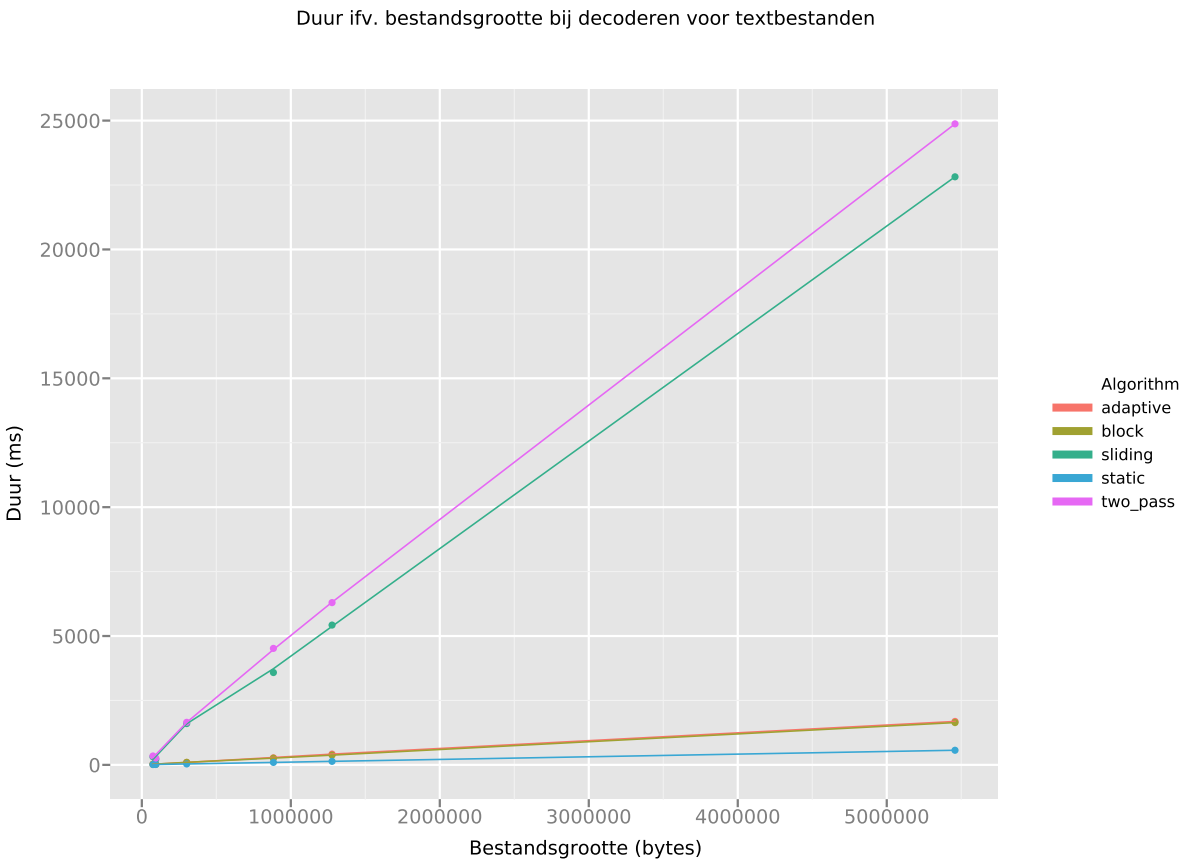


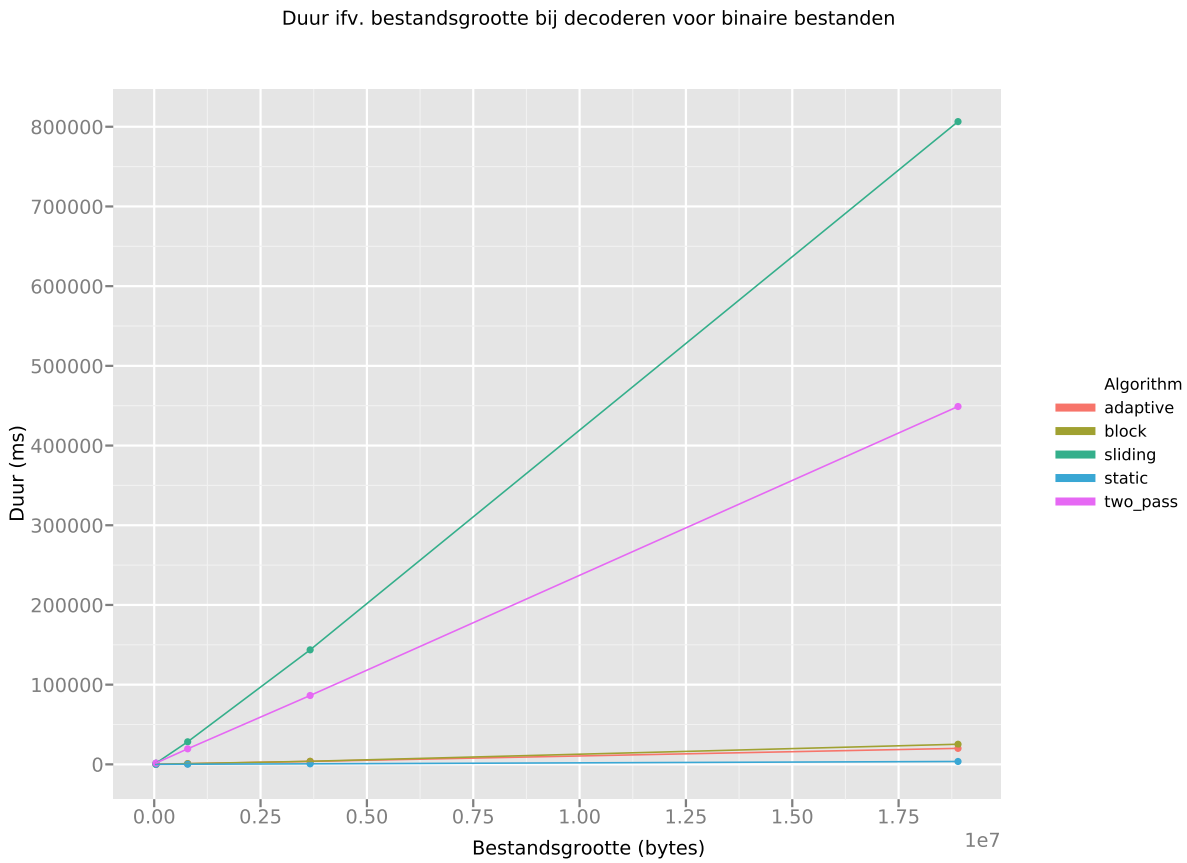
We zien ook dat het 'sliding window' algoritme sneller is dan het 'two pass' algoritme voor tekstbestanden, maar dat het omgekeerde geldt voor binaire bestanden. Dit is te verklaren door het feit dat er bij binaire bestanden veel meer willekeurige bytes zijn. Bij het gebruik van een sliding window kan het dus vaker voorkomen dat er een bepaalde byte meerdere keren uit het window (en dus de boom) verdwijnt. Terwijl we bij het 'two pass' algoritme zeker weten dat elke byte slechts 1 keer verwijderd zal worden.



4.1.2 Decoderen

Voor het decoderen zien we gelijkaardige resultaten.


Duur ifv. bestandsgrootte bij decoderen voor textbestanden
Duur (ms)
25000
20000
15000
10000
5000
0
0
1000000
2000000
3000000
4000000
5000000
Bestandsgrootte (bytes)
Algorithm
adaptive
block
sliding
static
two_pass

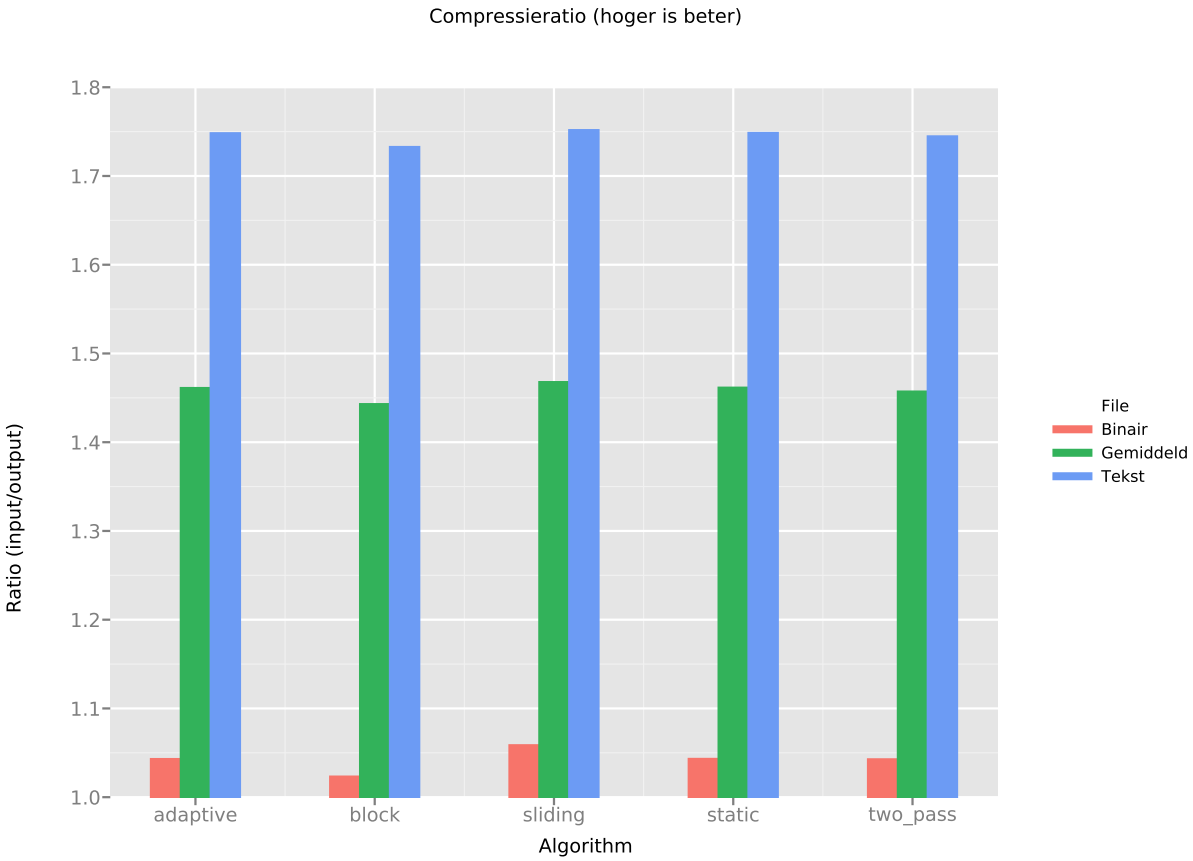


In alle gevallen is het 'statische' algoritme het snelste.

4.2 Compressiefactor

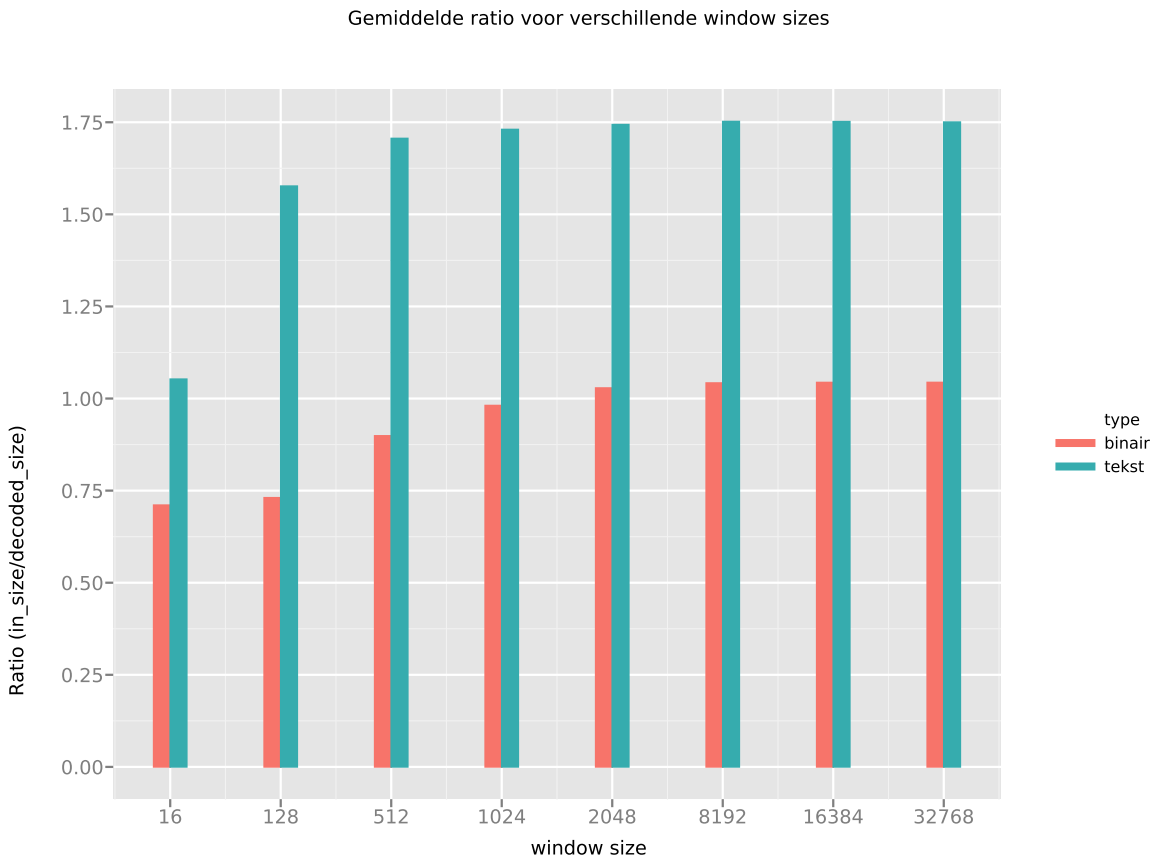
Voor de compressiefactor bekijken we de gemiddelde compressiefactor van de tekstbestanden, de binaire bestanden en het gemiddelde van die twee.

Hier zien we effectief dat binaire bestanden veel slechter encoderen dan tekstbestanden. Het 'sliding window' algoritme scoort het beste onder de binaire bestanden, vermoedelijk komt dit omdat het window ervoor zorgt dat de ruimtelijke lokaliteit van de binaire bestanden beter uitgebuit wordt.
Ook bij tekstbestanden scoort het 'sliding window' algoritme (nipt) het beste.

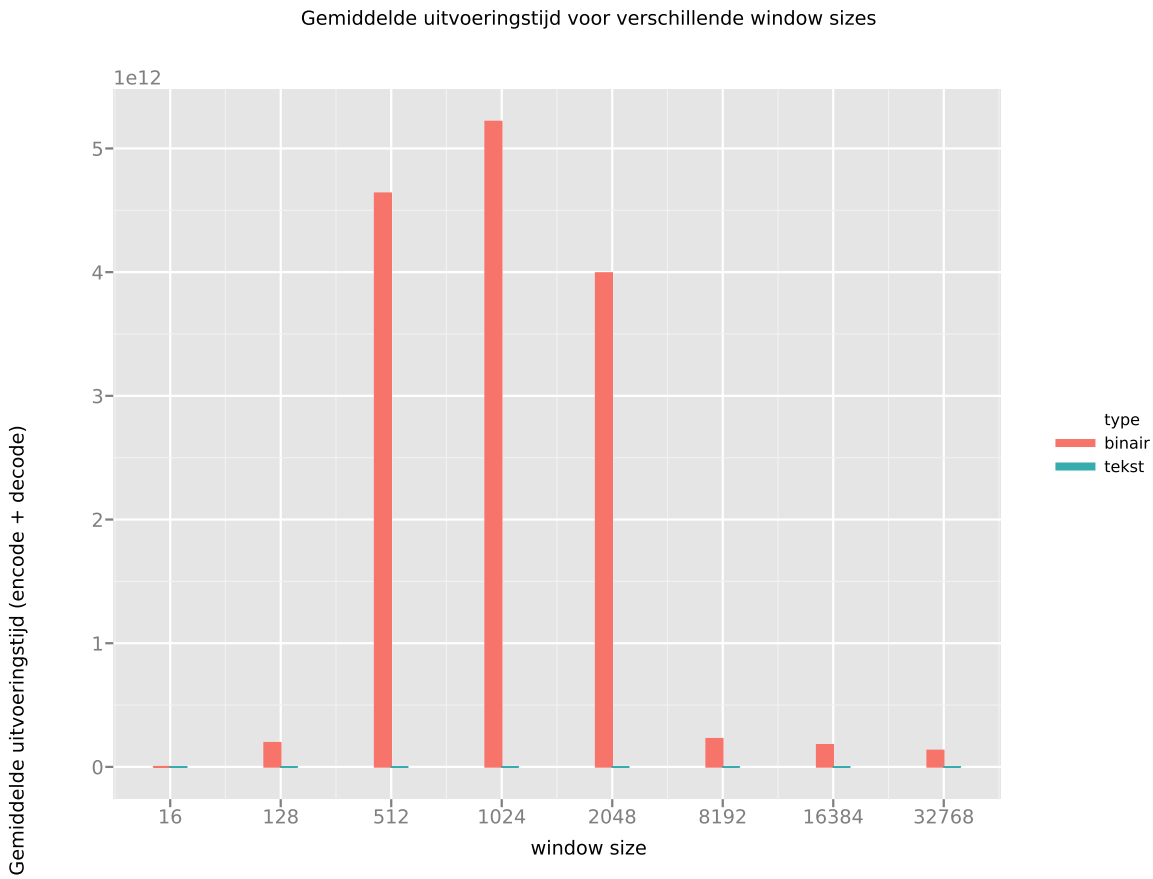


4.3 Windowgrootte

Om de ideale windowgrootte te vinden voor de gegeven datasets testen we het encoderen voor een aantal groottes uit.



We zien dat bij een window size van 8192 bytes de optimale compressieratio wordt bereikt voor zowel binaire als tekstbestanden. Kleine window sizes zorgen er zelfs voor dat het bestand groeit bij binaire bestanden omdat de overhead van het algoritme (het uitschrijven van nng) te groot wordt.



Hier merken we op dat bij een window size van 512, 1024 en 2048 bytes het encoderen van binaire bestanden enorm lang duurt. Dit komt vermoedelijk omdat bij het 35MB rockefeller.tiff bestand het window niet genoeg van de dataset in het window kan cachen waardoor er bytes voortdurend uit het window verwijderd worden en nadien opnieuw toegevoegd worden.

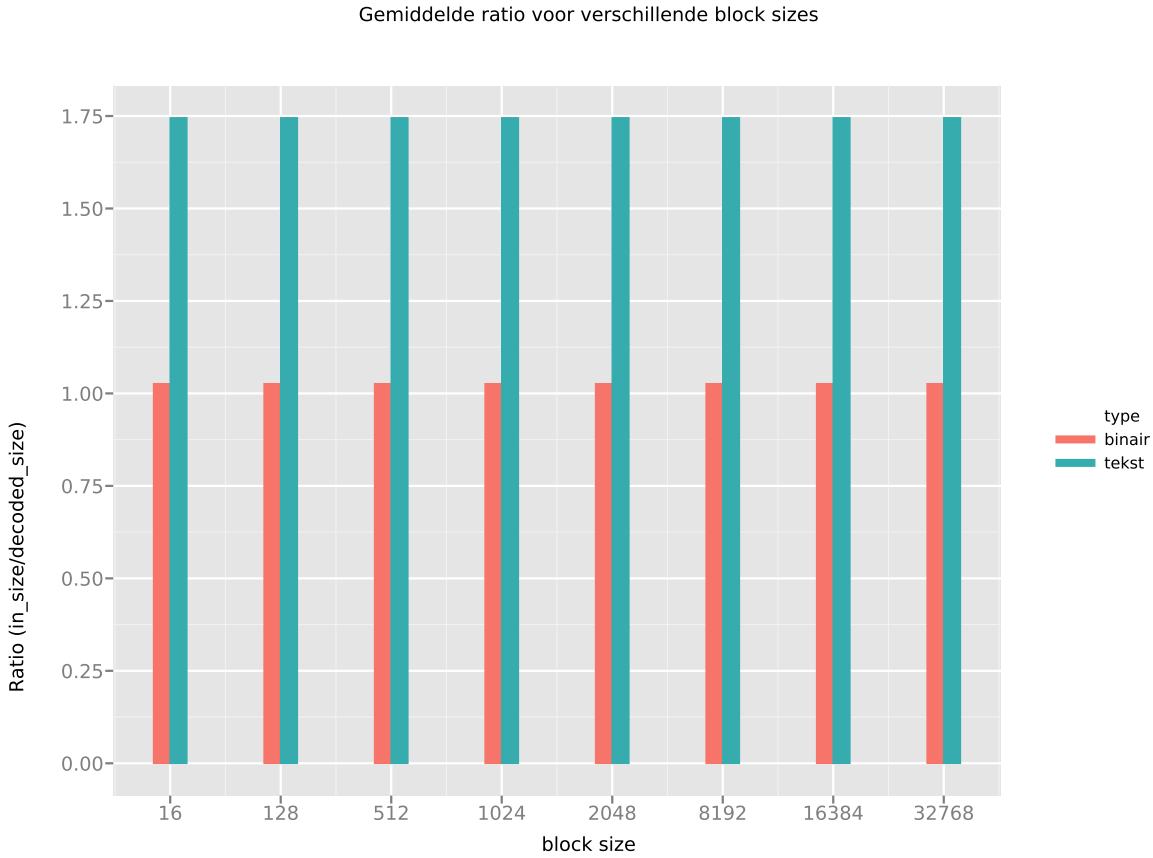
Bij window sizes van 16bytes en 128bytes vormt dit minder en probleem omdat de boom daar relatief klein blijft en de (dure) REDUCEFREQUENCY een stuk goedkoper is.

Vanaf 8192bytes zitten de bytes van de 'working set' van het bestand in het window. DELETE operaties komen dus minder vaak voor.

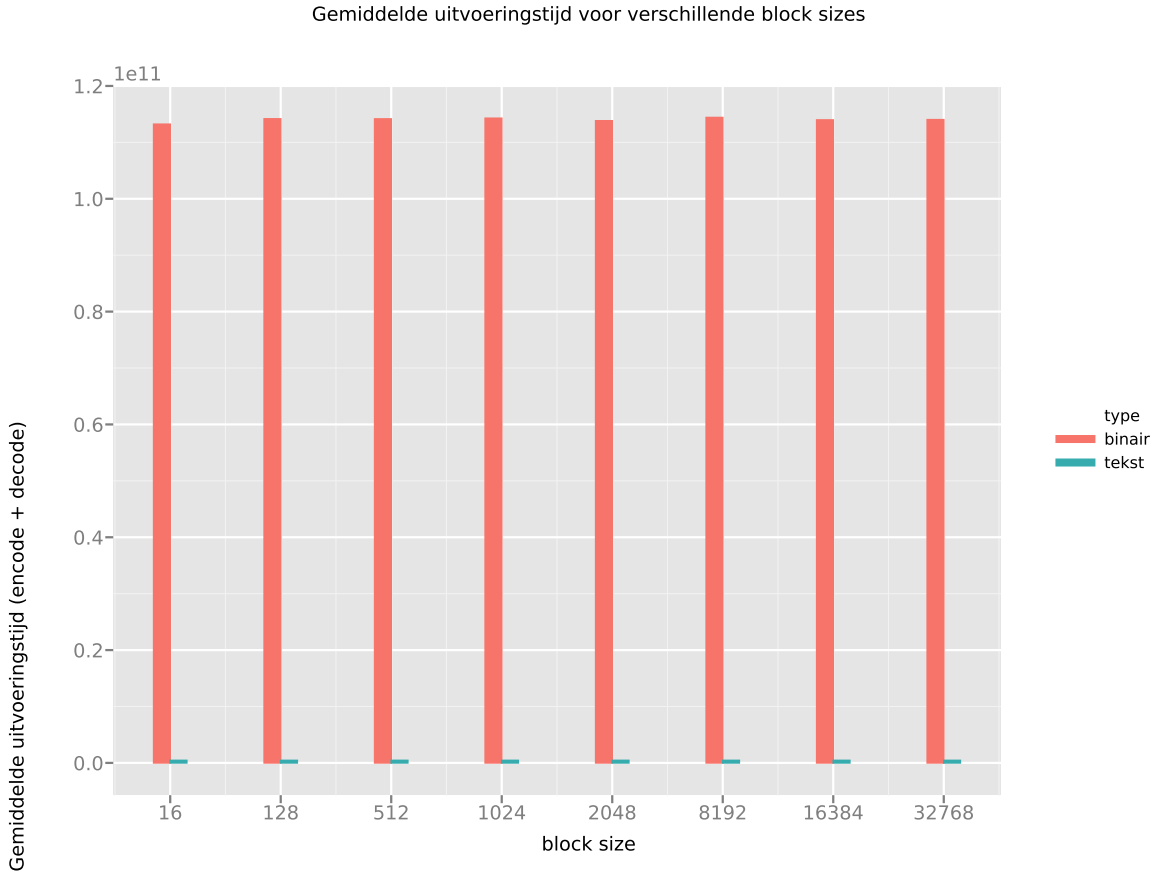
4.4 Blokgrootte

Analoog met hierboven kijken we welke blokgrootte voor de gegeven data ideaal is. Dit is niet helemaal hetzelfde als de ideale windowgrootte vinden aangezien er bij

elk nieuw blok een volledig nieuwe boom begonnen wordt. We willen dus zoveel mogelijk dezelfde data in een blok krijgen, maar het maakt niet uit waar de data binnen het blok zich bevindt.



We zien dat onafhankelijk van de blokgrootte de compressiefactor ongeveer hetzelfde blijft.

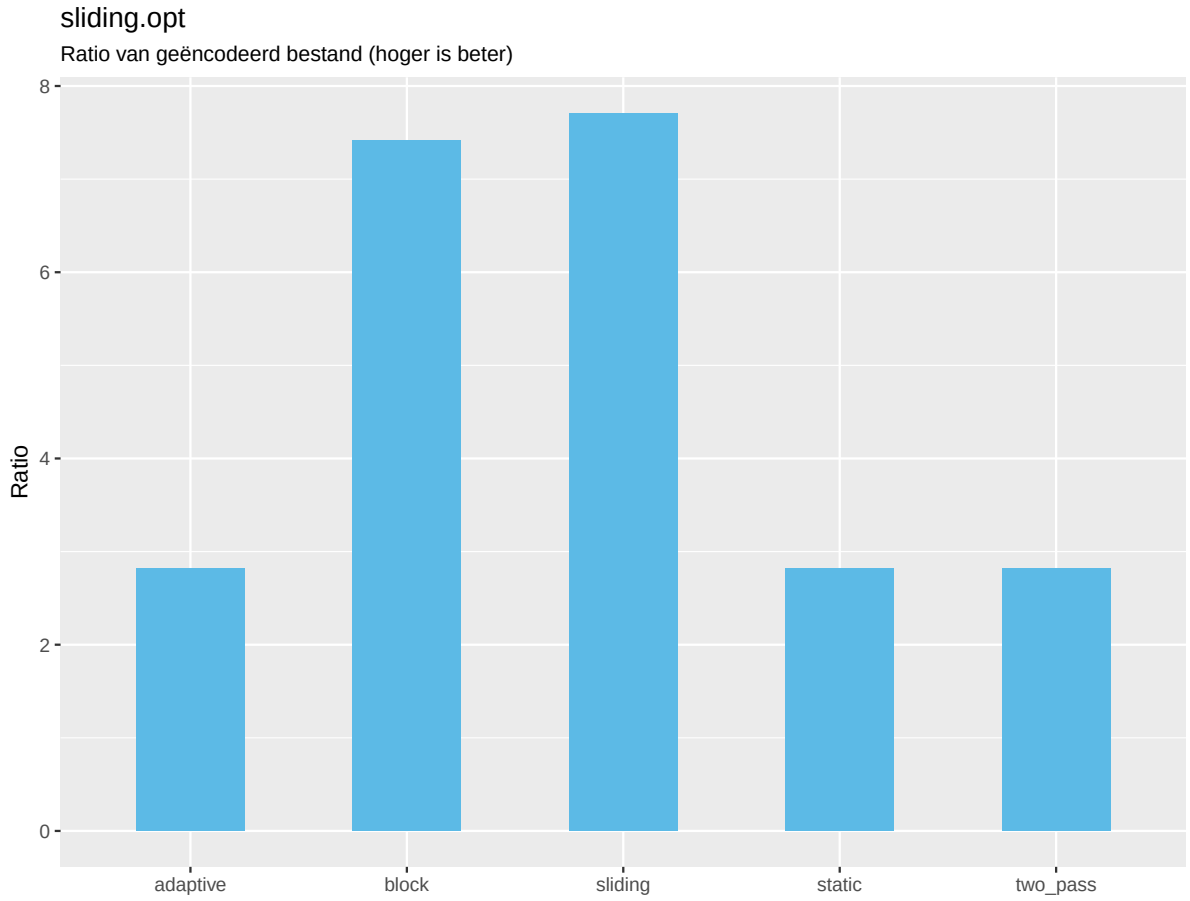


Ook het encoderen van binaire bestanden duurt langer dan het encoderen van tekstbestanden, maar ook hier merken we op dat onafhankelijk van de blokgrootte het encoderen ongeveer even lang duurt.

4.5 Optimale datasets

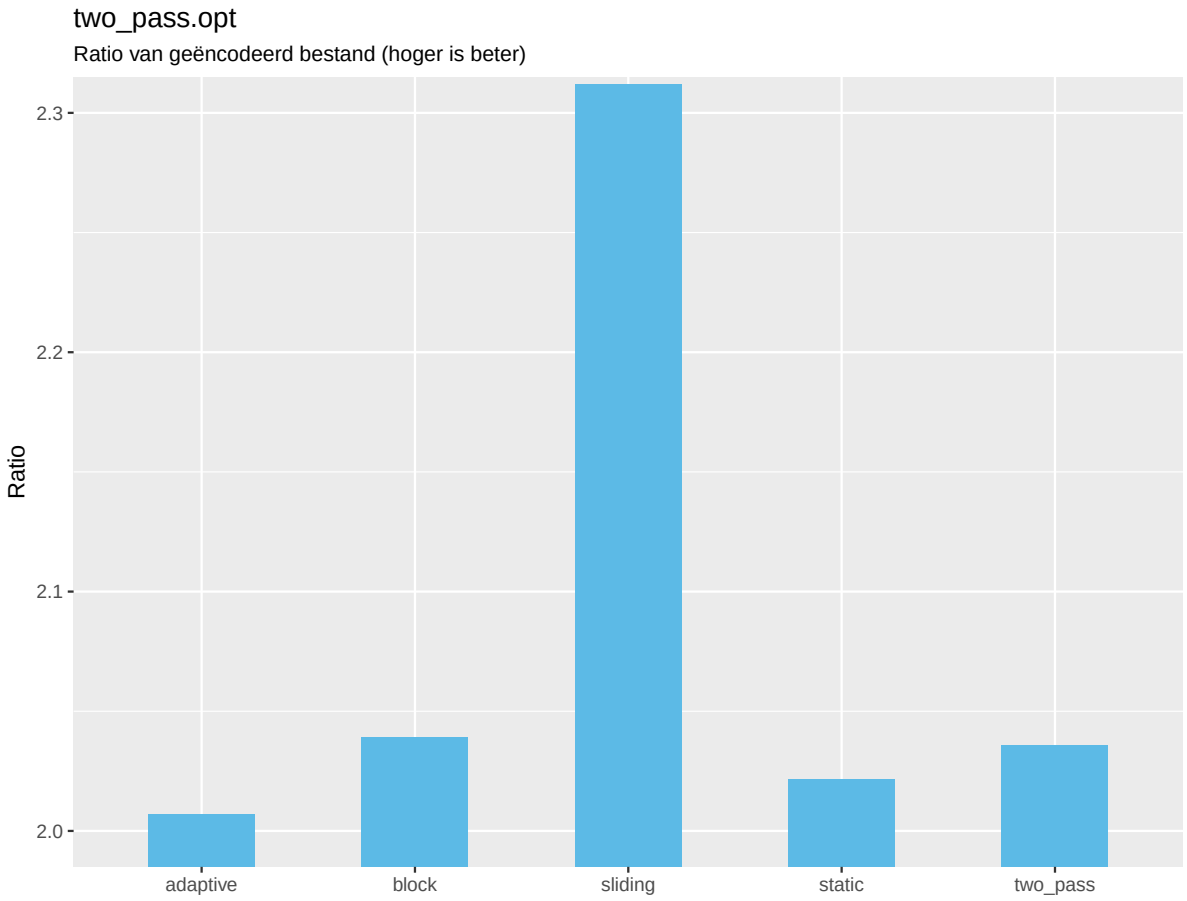
4.5.1 Sliding

We merken op dat het sliding algoritme voor dit soort datasets ongeveer 4x beter werkt dan de andere algoritmen. Ook zien we, zoals we eerder vermoedden, dat het bloksgewijze algoritme ook ongeveer even goede prestaties heeft.



4.5.2 Two pass

Bij deze dataset merken we op dat het verschil tussen de verschillende algoritmen erg klein is. Het sliding algoritme heeft hier de beste prestaties. Het two pass algoritme scoort wel net iets beter dan de andere algoritmen. Als we het bestand dus enkele ordes groter zullen maken zou dit wel een verschil kunnen maken.


two_pass.opt
Ratio van geëncodeerd bestand (hoger is beter)
Ratio
2.3
2.2
2.1
2.0
adaptive
block
sliding
static
two_pass

5. Besluit

5.1 Statische huffman

Het statische algoritme is in elk geval het snelste algoritme van alle geteste algoritmen. Dit is het grootste voordeel.

Een ander voordeel is dat statische huffman redelijk bestand is tegen transmissiefouten, indien er ergens een bitfout optreedt heeft dit minder dramatische gevolgen dan bij de andere algoritmen aangezien de boom niet dynamisch opgebouwd wordt.

Een nadeel van het statische huffman algoritme is echter het feit dat de tekst 2x doorlopen moet worden en online encoderen dus niet mogelijk is.

Een ander nadeel is dat de boom afzonderlijk moet opgeslagen worden en dit voor extra overhead kan zorgen in het bestand. In de praktijk hebben we echter waargenomen dat deze overhead ongeveer even groot is in vergelijking met de andere algoritmen (door de beperking op de alfabetgrootte en de overhead die andere algoritmen hebben).

5.2 Adaptive huffman

Het grootste voordeel van dit algoritme (alsook de volgende algoritmen) is dat de boom dynamisch wordt opgebouwd, dit algoritme is dus online.

Nog een voordeel is dat de boom niet afzonderlijk opgeslagen hoeft te worden en opgebouwd wordt naargelang de bytes komen (idem voor onderstaande algoritmen).

Een nadeel van dit algortime is dat de huffman-boom optimaal is voor de voorbije bytes en niet de komende bytes. In het geval dat een eerder

frequent voorkomende byte dus niet meer voorkomt, zal deze nog steeds relatief hoog in de boom blijven staan. Voor tekstbestanden komt dit geval echter niet vaak voor.

Ook is dit algoritme (en de volgende algoritmen) erg gevoelig voor bitfouten. 1 fout kan ervoor zorgen dat de rest van de tekst volledig ondecodeerbaar wordt.

5.3 Adaptive huffman met sliding window

Deze boom is nog altijd optimaal voor de voorbije bytes, maar het 'frame of reference' van het sliding window algoritme een stuk kleiner, waardoor de informatie in het window vaak nog relevant is voor de komende bytes.

Voor binaire bestanden kunnen we de huffman-boom als een soort cache beschouwen. Net als bij caches zal een kleinere boom de byte efficiënter kunnen coderen maar een cache misser zal altijd duur zijn. Indien we een goede balans van de cachegrootte kunnen vinden waar de window size groot genoeg is om ruimtelijke lokaliteit uit te buiten, maar het window niet zo groot is dat de huffman-boom te groot wordt, dan zullen we een optimale window size bereiken.

Een ander voordeel van het sliding window algoritme is dat er bytes uit de boom kunnen verdwijnen, de codes zullen dus over het algemeen korter zijn dan wanneer dit niet kan.

Het grootste nadeel van dit algoritme is dat dit algoritme relatief traag is door de REDUCEFREQUENCY operatie. Indien een suboptimale window size wordt gekozen kan dit het algoritme zelfs vele malen trager maken omdat er continu bytes verwijderd en opnieuw toegevoegd worden.

5.4 Two pass adaptive huffman

Het grootste voordeel van two pass adaptive huffman is dat de boom optimaal is voor de bytes die nog moeten volgen.
In de praktijk zal dit echter niet veel verschil maken omdat bij tekstbestanden de voorbije bytes een goede referentie vormen voor de bytes die nog moeten komen. Bij binaire bestanden buit het 'sliding window' algoritme dan weer de lokaliteit beter uit, waardoor 'two pass' ook hier slechter(/even goed) scoort dan de andere algoritmen.

Een nadeel is dat dit algoritme ook weer relatief traag is, dit omdat we weten dat elke character uit de boom zal verwijderd moeten worden.

Hier moet de tekst 2x doorlopen worden bij het encoderen. Maar bij het decoderen is dit niet noodzakelijk (we moeten niet verder kijken om te zien hoeveel bits we op het einde moeten negeren).

5.5 Bloksgewijze adaptive huffman

Het bloksgewijze algoritme presteert qua compressie net iets minder goed dan de andere algoritmen. Maar daartegen staat dat het algoritme erg schaalbaar is, in principe zouden er een erg groot aantal symbolen (van meer dan 1byte bijvoorbeeld) en een oneindig grote dataset kunnen gebruikt worden en zou dit algoritme nooit problemen met het geheugen krijgen, een luxe die er bij statische huffman bvb. niet is.

In theorie kan dit algoritme ook sneller werken dan adaptive huffman indien er heel veel verschillende bytes voorkomen en de blokgrootte zo gekozen wordt dat grote huffman-bomen vermeden worden.

Dit algoritme werkt ook ongeveer met hetzelfde principe als het 'sliding window' algoritme (met een ander doel), echter is het veel sneller omdat na elke blok de boom volledig opnieuw opgebouwd wordt en de characters niet telkens verwijderd moeten worden. Natuurlijk is er hierdoor wel wat meer overhead.

Wanneer we headers zouden gebruiken tussen de verschillende blokken zouden we er zelfs voor kunnen zorgen dat een bitfout slechts 1 blok corrupt maakt. Wat 'bloksgewijze adaptieve huffman met headers' dus een practischer algoritme zou maken dan het gewone 'adaptive huffman' algoritme als de data over een kanaal met veel ruis verstuurd zou moeten worden.
Ook kunnen we wanneer we headers gebruiken de tekst parallel encoderen en decoderen indien we nog enkele wijzigingen aan het algoritme zouden maken.

Wanneer we de blokgrootte groter maken zal de overhead per blok lager liggen omdat er in totaal minder vaak een 'nng' zal voorkomen, natuurlijk verliezen we vanaf een bepaalde blokgrootte de hiervoor genoemde voordelen.

6. Wijzigingen aan de code sinds het indienen op 1/12

- Er ontbrak nog een cast naar **unsigned char** bij het statische huffman algoritme. Dit had als gevolg dat bij het encoderen van binaire bestanden er segfaults konden optreden. Bij ASCII bestanden kwamen deze fouten niet voor omdat de char waarde tussen 0 en 127 ligt en dus niet negatief gaat.
- Om verschillende blok/window-grootten te kunnen testen werd de '-s' parameter toegevoegd waarmee een custom blok/window size kan meegegeven kan worden die dan de standaardgrootte zal overschrijven (1024).