Analisis algoritma Adaptive Huffman Coding

Rizqy Faishal Tanjung - 1406622856

Fakultas Ilmu Komputer, Universitas Indonesia

rizqyfaishal27@gmail.com

Abstrak

Salah satu algoritma *lossless compression* yang cukup sukses adalah Huffman Code. Huffman code menggunakan pendekatan *greedy* untuk mengubah simbol-simbol yang sering muncul dalam data menjadi seringkas mungkin sehingga menghasilkan compression ratio yang cukup tinggi. Huffman code menggunakan *two-pass step*, yaitu membaca keseluruhan data untuk membangun model tree, dan mengubah data tersebut ke bentuk deretan *prefix code*. Hal tersebut memerlukan memori yang cukup besar untuk menampung sementara data yang akan dibaca jika input datanya besar. Adaptive Huffman code mengubah *two-pass* proses *encoding* nya menjadi *one-pass*, sehingga setiap membaca aliran karakter akan langsung di konversi menjadi *prefix code* dan tidak memerlukan memori tambahan. Komputer dengan kapasitas *low-memory* bisa menggunakan teknik Huffman untuk mengkonversi data dan sangat cocok dalam kasus *data-streaming* yang umumnya terdapat pada internet.

Kata kunci: adaptive, compression, low-memory, data-streaming, huffman code

Latar Belakang

1. Huffman code harus membaca semua data yang akan di *encoding* terlebih dahulu sebelum membuat model encoder nya (tree)
2. Huffman kurang cocok digunakan untuk aplikasi yang streaming data
3. Huffman code memerlukan memory tambahan untuk menampung data yang akan di encode

Pendahuluan

Data compression merepresentasikan suatu data ke dalam bentuk yang lebih ringkas atau pendek dalam suatu ukuran besaran memori dengan cara mengidentifikasi struktur dari data tersebut. Algoritma data compression dibagi menjadi dua bagian yaitu *lossless* dan *lossy* compression. *Lossless compression* tidak menghilangkan informasi dari data sehingga data yang sudah terkompres dapat dikembalikan lagi ke dalam kondisi data asli yang sesungguhnya. *Lossy compression* mengizinkan untuk menghilangkan informasi dari data yang asli untuk menghasilkan performa yang maksimal, biasanya data yang sudah terkompresi menggunakan algoritma *lossy compression* tidak dapat direkonstruksi lagi ke dalam bentuk data aslinya. Pada bahasan ini hanya membahas algoritma *lossless compression*. Dalam *lossless compression* meliputi dua tahap yaitu encoding dan decoding. *Encoding* merupakan proses transformasi dari data asli ke data yang sudah dalam bentuk terkompresi. *Decoding* merupakan proses kebalikan dari encoding. Salah satu algoritma *lossless* yang cukup terkenal yaitu *huffman coding*.

Pengukuran performa algoritma data compression

Performa yang akan diukur meliputi 3 hal yaitu:

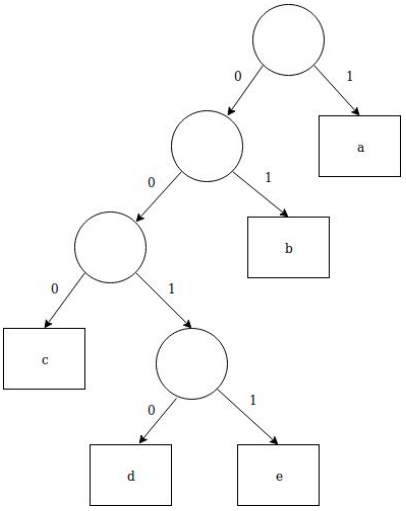
- *compression* ratio: perbandingan antara ukuran data yang sudah terkompresi dengan data aslinya
- *running time* proses *encoding*
- *running time* proses *decoding*
- memori yang digunakan dalam proses *encoding* dan *decoding*

Huffman coding

Huffman menggunakan informasi statistik setiap bagian kecil dari data atau biasanya karakter dalam teks, informasi distribusi karakter tersebut digunakan untuk membangun model yang menghasilkan *prefix codes. Prefix codes* merupakan *codes* dimana suatu codeword bukan merupakan *prefix* dari *codeword* yang lain. Untuk menghasilkan *prefix code*s dan *compression* ratio yang optimal terdapat 2 kondisi yang harus dipenuhi dalam Huffman codes yaitu:

1. Karakter yang frekuensi kemunculannya besar akan mempunyai *codeword* yang pendek
2. Dua simbol yang sama-sama frekuensi kemunculannya kecil akan mempunyai panjang *codewords* yang sama.

Huffman code menggunakan struktur data tree dimana setiap *leaf* nya akan menentukan karakter yang akan di *encode* dan jalur untuk menuju *leaf* tersebut merupakan hasil dari proses *encode* tersebut. Sebagai contoh:



Gambar 1. Struktur data tree digunakan dalam Huffman codes

Setiap kemunculan:

- karakter “a” akan diganti dengan binary “1”
- karakter “b” akan diganti dengan binary “01”
- karakter “c” akan diganti dengan binary “000”
- karakter “d” akan diganti dengan binary “0010”
- karakter “e” akan diganti dengan binary “0011”

Misalkan urutan karakter “bcabaacdea”, akan diubah menjadi urutan binary “01000101110000010000111”. Jika dilakukan perhitungan memori yang dibutuhkan urutan karakter di atas jika dalam ascii akan memerlukan memori sebesar 10 x 8 bit = 80 bit, sedangkan dalam binary hanya akan memerlukan memori sebesar 22 x 1 bit = 22 bit. Sehingga compression ratio hampir mencapai 75%.

Huffman code memerlukan informasi statistik setiap karakter dari sumber datanya. Sehingga, untuk melakukan proses encoding data harus dibaca secara keseluruhan, dihitung informasi statistik, setelah itu baru melakukan proses encoding. Dalam kasus streaming data, keseluruhan data harus disimpan dalam buffer terlebih dahulu sebelum dilakukan proses encoding, sehingga perlu tambahan memory atau storage. Terdapat varian dari Huffman code yaitu melakukan *update* informasi *tree* di setiap karakter yang akan di proses, algoritma tersebut adalah Adaptive Huffman Code.

HuffmanCodeEncode(D)	Cost	Times
1. define C, is array for counting simbol	c1	1
2. for i = 1 to LENGTH(D)	c2	n + 1
3. C[D[i]] += 1	c3	n
4. define Q, is Heap to store node	c4	1
5. for j = 1 to LENGTH(C)	c5	c + 1
6. build node from C[j] and add to Q	c6	c
7. while LENGTH(Q) > 1	c7	c
8. node left = delMin(Q)	c8	c - 1
9. node right = delMin(Q)	c9	c - 1
10. add combiner left and right node to Q	c10	c - 1
11. node root = take first element from Q	c11	1
12. for i = k to length(D)	c12	n + 1
13. traverse D[i] from root node to leaf to get prefix codes	c13	n

cost c6, c8, c9, c10 = log (length(Q))
cost c13 = log c

T(n) = c1 + c2(n+1) + c3n + c4 + c5(c+1) + c6*c + c7*c + (c8+c9+c10)(c-1) + c11 + c12(n+1) + c13*n

Sehingga jika diuraikan **T(n) = O(n*log(c))**

dimana n adalah jumlah input simbol dan c jumlah *distinct* simbol.

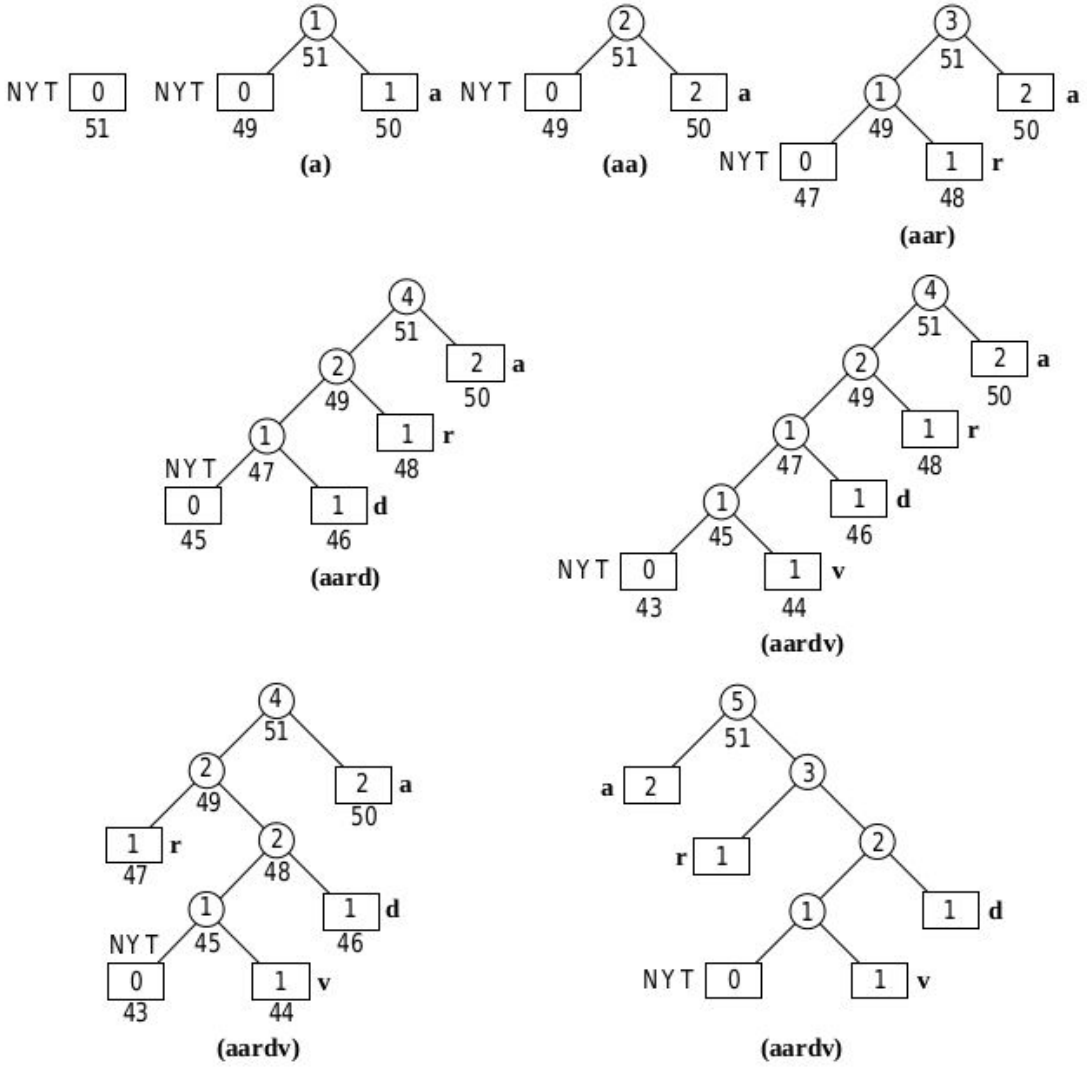
Adaptive Huffman Coding

Adaptive Huffman code melakukan proses *encoding* hanya memerlukan one-pass, jadi proses mengumpulkan informasi probabilitas dari karakternya dilakukan secara inkremental, dan akan terus diupdate sampai tidak ada karakter yang tersisa. Hal ini cukup menguntungkan karena tidak perlu memori tambahan untuk menampung semua keseluruhan data untuk menghitung informasi statistiknya (one-pass).

Jika dalam Huffman code proses pembentukan tree hanya dilakukan sekali dan dipakai seterusnya untuk proses *encoding* jadi tidak ada prosedur *update tree*, beda halnya dalam pendekatan adaptive. Setiap satu simbol selesai dilakukan proses *encoding*, maka *tree* akan di *update* supaya menjaga konsistensi pada proses decoding. Dalam pendekatan *adaptive* juga tidak dilakukan penghitungan statistik setiap simbol, dan hal inilah yang membuat pendekatan adaptive tidak perlu memori tambahan. Setiap pada proses encoding simbol, setiap simbol yang *unique* akan dibuat dua buah node, sehingga bila di keseluruhan data terdapat 26 simbol maka akan ada 52 node. Jadi di awal konfigurasi dengan adaptive harus ditentukan juga jumlah maksimal simbol *unique* yang akan dipakai. Node akan mempunyai informasi tambahan yaitu *weight* dan *node number*. Terdapat 1 node spesial yaitu NYT node untuk menandai pelacakan jalur pada proses decode. NYT node akan selalu mempunyai *node number* terendah dan berposisi paling bawah kiri dalam *tree*. Setiap node mempunyai ketentuan sebagai berikut:

1. Setiap *node* yang bukan *leaf* node harus mempunyai dua *siblings*
2. *Sibling* sebelah kiri akan mempunyai *weight* yang lebih rendah dibandingkan dengan yang kanan

Contoh misalkan string “aardv” jika dilakukan encoding menggunakan Adaptive Huffman Code dengan catatan akan menggunakan maksimal 26 unique simbol.



Gambar 1. Proses pembentukan tree untuk string “aardv” [2]

Tree yang terbentuk di setiap simbol yang dibaca akan berbeda-beda karena setelah proses *encoding* akan selalu melakukan prosedur *update tree*. Prosedur *update tree* tersebut memungkinkan setiap node akan selalu mempunyai 2 ciri yang terdapat diatas tadi. Tree yang terbentuk hampir menyerupai tree yang linear, sehingga jika semakin banyak simbol yang unique yang dipakai maka running time untuk traversal ke external node hampir mencapai c (jumlah simbol *unique*). Tree terbentuk hampir mendekati linear karena setiap penambahan simbol yang belum dihitung dan ditambahkan ke tree, selalu ditempatkan pada bagian NYT node, sehingga setiap simbol akan menambah 1 depth pada tree, serta proses swapping node dilakukan hanya pada level tree yang sama (jadi node pada level 5 tidak bisa di swap dengan node pada level 4).

Kompleksitas untuk prose encoding Adaptive Huffman Code:

AdaptiveHuffmanCodeEncode(D)	Cost	Times

1. define C, is hash table for checking a simbol have been computed	c1	1
2. for i = 1 to LENGTH(D)	c2	n + 1
3. if D[i] is in hashtable	c3	n
4. increment weight of node D[i]	c4	n
5. traverse prefix codes from tree	c5	n
6. update procedure	c6	n * c
7. else	c7	n
8. node newNode = create new node for D[i]	c8	n
9. add newNode to hashtable	c9	n
10. add newNode to tree	c10	n
11. update procedure	c11	n * c

Prosedur update tree menjadi komponen utama dalam kompleksitas Adaptive Huffman Code. Karena tree hampir berbentuk *linear* maka sehingga proses *update tree* memerlukan iterasi sejumlah c (jumlah simbol unique yang sudah dihitung).

$T(n) = c1 + c2(n+1) + (c3+c4+c5+c7+c8+c9+c10)n+ (c6+c11)*nc$

$T(n) = O(n*c)$

Algoritma Vitter

Terdapat beberapa optimisasi supaya bentuk tree tidak mendekati linear yaitu algoritma vitter pada proses update tree nya. Satu ketentuan tambahan untuk setiap node yaitu:

1. Dari kiri ke kanan dan bawah ke atas *weight* dari setiap *node* akan bertambah

Algoritma vitter bertujuan supaya bentuk *node* bisa lebih seimbang dan proses *update tree* bisa memakan waktu log(c) di setiap iterasi data. Sehingga :

$T(n) = O(n*log(c))$

Hasil Eksperimen

Spesifikasi Hardware untuk testing

1. Intel core i3 1.8GHz
2. RAM 4gb
3. Operating system: linux Ubuntu 18.10

Percobaan pertama dengan menggunakan **56 bytes** dataset dengan varian karakter yang sedikit.

Nama Algoritma	Encoding Time	Decoding Time	Compression Ratio
Huffman Code	0 ms - 0 MB	1 ms - 0 MB	69.4 %
Adaptive Huffman Code	1 ms - 0 MB	0 ms - 0 MB	60.7 %
Adaptive Huffman Code (Vitter)	1 ms - 0 MB	0 ms - 0 MB	60.7 %

Percobaan kedua dengan menggunakan dataset **583392 bytes.**

Nama Algoritma	Encoding Time / memory	Decoding Time / memory	Compression Ratio
Huffman Code	226 ms - 10 MB	257 ms - 6 MB	47.6 %
Adaptive Huffman Code	547 ms - 44 MB	317 ms - 6 MB	-12 %
Adaptive Huffman Code (Vitter)	589 ms - 30 MB	354 ms - 4 MB	- 12 %

Percobaan ketiga dengan menggunakan dataset **5079398 bytes**.

Nama Algoritma	Encoding Time	Decoding Time	Compression Ratio
Huffman Code	1214 ms - 45 MB	1968 ms - 7 MB	45 %
Adaptive Huffman Code	3721 ms - 45 MB	2917 ms - 7 MB	-31 %
Adaptive Huffman Code (Vitter)	3616 ms - 40 MB	3320 ms - 7 MB	- 31 %

Percobaan keempat dengan menggunakan dataset **67565846 bytes**.

Nama Algoritma	Encoding Time / memory	Decoding Time	Compression Ratio
Huffman Code	14906 ms - 167 MB	17157 ms - 8MB	46.6 %
Adaptive Huffman Code	54886 ms - 40MB	45395 ms - 7MB	-12 %

Adaptive Huffman Code (Vitter)	52752 ms - 37MB	44713 ms - 7MB	- 12 %

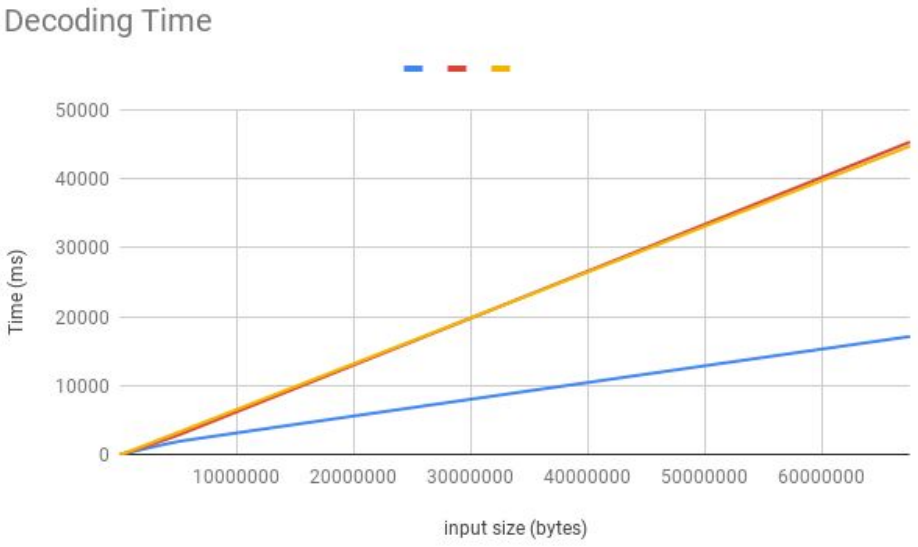
Percobaan keempat dengan menggunakan **67565846 bytes** namun dengan batasan memory 256 MB yang digunakan.

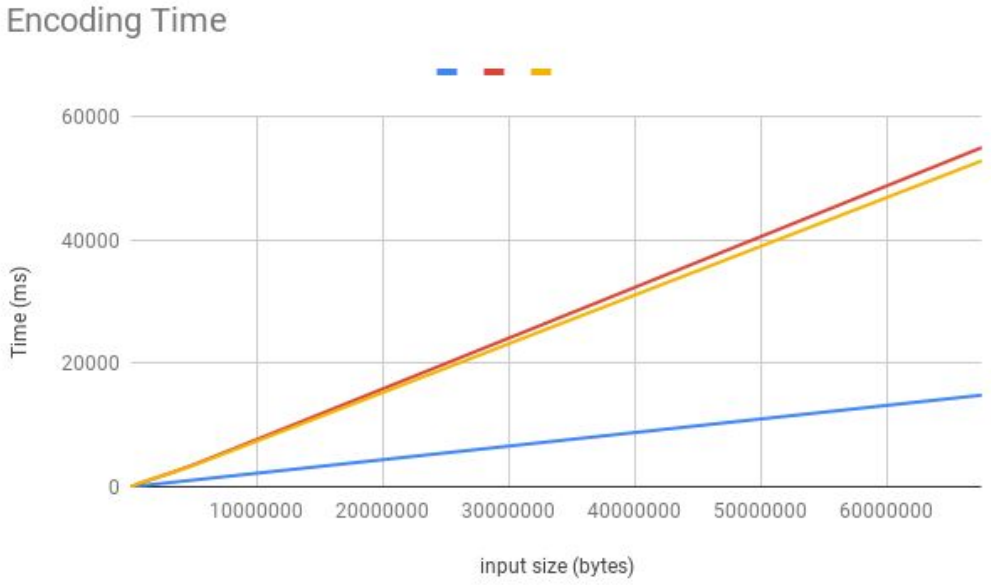
Nama Algoritma	**Encoding Time / memory**	**Decoding Time**	**Compression Ratio**
Huffman Code	out of memory error	16069 ms - 8MB	46.6 %
Adaptive Huffman Code	56250 ms - 35MB	45950 ms - 7MB	-12 %
Adaptive Huffman Code (Vitter)	55534 ms - 37MB	40761 ms - 6MB	- 12 %

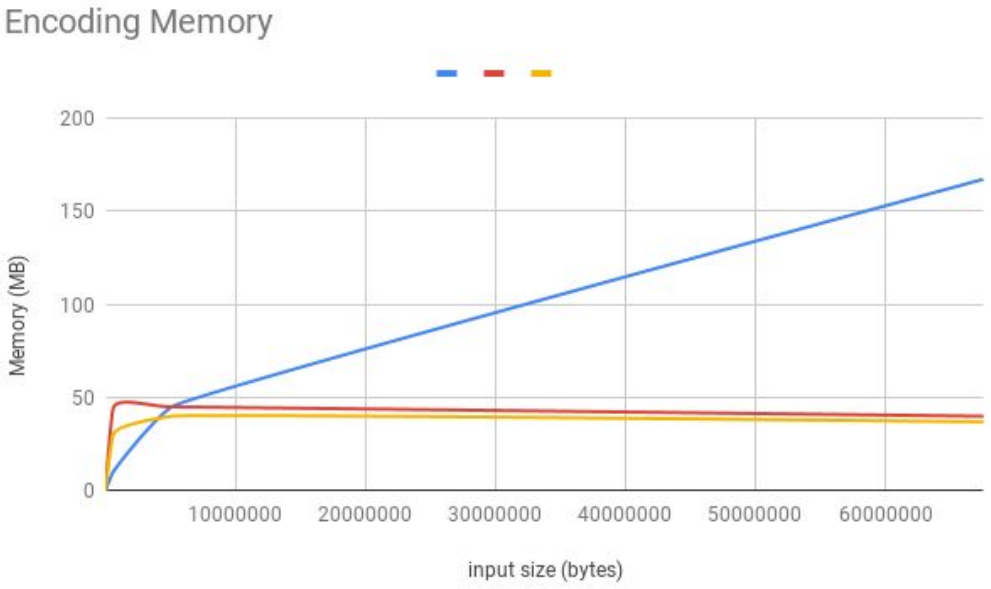
Percobaan keempat dengan menggunakan **67565872 bytes** namun dengan batasan variasi karakter (7) yang digunakan.

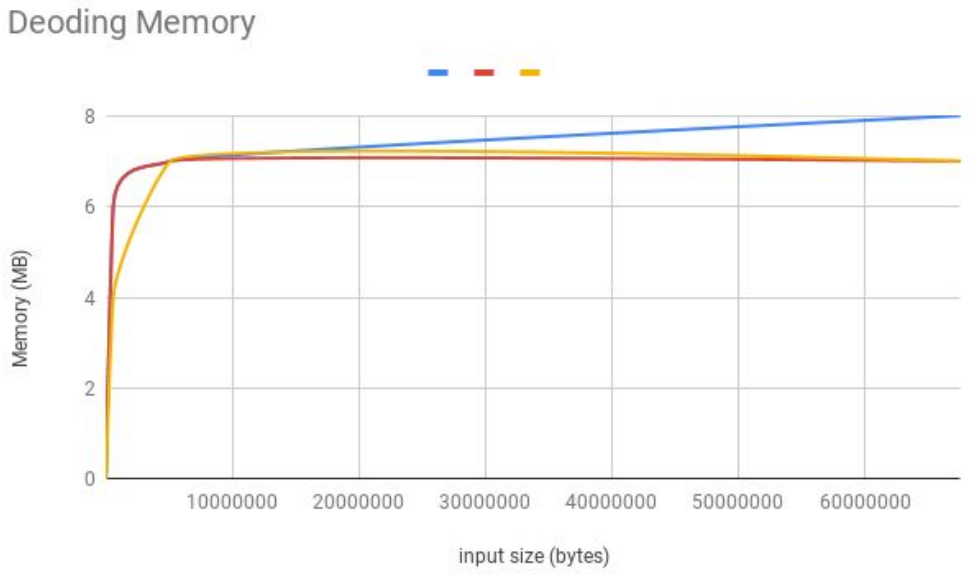
Nama Algoritma	**Encoding Time**	**Decoding Time**	**Compression Ratio**
Huffman Code	14607 ms - 288MB	14461 ms - 10MB	62.5 %
Adaptive Huffman Code	27886 ms - 40MB	18056 ms - 7MB	50 %
Adaptive Huffman Code (Vitter)	30532 ms - 37MB	16126 ms - 8MB	50 %

Analisis


Decoding Time
50000
40000
30000
20000
10000
0
Time (ms)
10000000
20000000
30000000
40000000
50000000
60000000
input size (bytes)


Encoding Time
60000
40000
20000
0
Time (ms)
10000000
20000000
30000000
40000000
50000000
60000000
input size (bytes)





Note:

- Biru: Huffman Code
- Merah: Adaptive Huffman Code
- Kuning: Adaptive Huffman Code (Vitter)

Dilihat dari grafik perbedaan yang mencolok terjadi pada proses *encoding*, *running time* non-adaptive huffman code lebih cepat dibandingkan dengan yang pendekatan adaptive hal itu terjadi seperti pada penjelasan diatas, proses encoding pada non-adaptive Huffman code tidak memerlukan prosedur *update tree* di setiap karakter yang dibaca sehingga lebih cepat dan bentuk tree yang dihasilkan Huffman code (tergantung inputnya) biasanya tidak berbentuk mendekati linear sehingga *traverse code* dari *root* hingga *leaf* bisa dicapai dengan waktu log(c) per karakter dibandingkan dengan c pada pendekatan adaptive. Pada non-adaptive Huffman *code* sendiri jumlah *node* adalah jumlah karakter distinct

dibandingkan dengan 2 kali jumlah karakter pada pendekatan adaptive. Sehingga *node* yang ditelusuri jauh lebih sedikit.

Memori yang digunakan adaptive bersifat konstan berbeda dibandingkan dengan non-adaptive yang linear. Hal ini terjadi karena non-adaptive akan menampung data sementara untuk membuat tree dan melakukan encode di setiap simbol, sedangkan di bagian adaptive, memori hanya diperlukan untuk menampung simbol (memakai buffer sebanyak 8192 bytes) yang tepat akan diproses dan setelah itu akan dibebaskan memori tersebut dan digunakan kembali untuk simbol berikutnya.

Kesimpulan

Tidak ada algoritma yang dapat *one-fits-all,* tergantung dengan kebutuhan dan jenis kasus yang akan diselesaikan. Dari hasil eksperimen diatas, Huffman code dalam hal c*ompression ratio* dan *running time* memang lebih bagus dibandingkan dengan pendekatan adaptive. Namun, pada komputer yang memiliki keterbatasan memori akan sangat tidak berguna untuk memproses jumlah input yang besar. Sebaliknya, adaptive huffman code memerlukan sedikit memori karena pada saat setiap simbol dibaca akan langsung diubah menjadi prefix code.

Code references:

- https://github.com/rizqyfaishal27/paper-daa
- http://ben-tanen.com/adaptive-huffman/

References:

[1] Dath, D., & Panicker, J. V. (2017, April). Enhancing adaptive huffman coding through word by word compression for textual data. In *2017 International Conference on Communication and Signal Processing (ICCSP)* (pp. 1048-1051). IEEE.

[2] Sayood, K. (2017). *Introduction to data compression*. Morgan Kaufmann.