Algoritma Huffman dan Kompresi Data

David Soendoro ~ NIM 13507086
Jurusan Teknik Informatika ITB, Bandung, email: if17086@students.if.itb.ac.id

Abstract *– Algoritma Huffman merupakan salah satu algoritma yang diajarkan pada mata kuliah struktur Diskrit (IF2091) yang membantu kita dalam menangani data dengan ukuran yang besar. Termasuk dalam penanganan format mp3 dan jpg. Tetapi yang paling menarik dari algoritma Huffman adalah kemampuan algoritma tersebut mengompresi data. Adapun algoritma Huffman memiliki 2 fase utama yaitu pengubahan menjadi kode (encoding), dan mengembalikan kode (decoding). Algoritma Huffman yang telah dipelajari adalah Algoritma Huffman Statis yang memiliki kompleksitas O (n log n) –dengan n adalah bit– yang sangat ringan, Algoritma ini berkembang menjadi Algoritma Huffman Dinamis yang menarik karena menjadi algoritma yang lebih efisien dan mampu mengompresi data lebih cepat serta lebih mampat dengan kompleksitas O (log n).*

Kata Kunci*: algoritma Huffman statis, algoritma Huffman dinamis, encoding, decoding, kompresi data.*

1. PENDAHULUAN

Algoritma Huffman merupakan algoritma dasar pengompresian, algoritma tersebut merupakan algoritma yang paling sederhana dibandingkan dengan algoritma pengompresian lainnya. Algoritma Huffman Statis hanya membutuhkan 2 langkah dalam pengompresian hal ini membuatnya begitu cepat, yaitu encoding dan decoding karena peta kode akan selalu sama. Akan tetapi, karena selalu menggunakan peta kode yang sama maka seringkali algoritma Huffman (statis) dianggap tidak efisien, apalagi pada dunia modern ini di mana proses dinamik yang dilakukan internet seperti streaming audio dan video kerap dilakukan.

Namun karena pentingnya algoritma Huffman sebagai landasan mempelajari algoritma pengompresian selanjutnya untuk itulah makalah ini dibuat. Algoritma Huffman merupakan dasar bagi algoritma pengompresian yang kerap digunakan sekarang ini. Bahkan pengompresian .Zip masih menggunakan algoritma Huffman yang membuat .Zip lebih cepat dibuat namun memiliki size yang lebih besar (karena seperti dikatakan di atas algoritma Huffman cepat namun tidak efisien).

Juga akan ada penjelasan tentang perkembangan dari algoritma Huffman Statis, yaitu algoritma Huffman Dinamis yang lebih efisien, Penulis bertujuan menerangkan perbedaan kedua algoritma tersebut masing-masing kelebihan dan kekurangannya.

Untuk mencapai tujuan tersebut di atas maka penulis telah melakukan pembelajaran dalam pencarian lebih lanjut informasi tentang algoritma Huffman statis dan algoritma Huffman dinamis dalam pengompresian data.

2. Kompresi Data dan Hubungannya dengan Algoritma Huffman

2.1. Kompresi Data

Kompresi secara harafiah berarti mengecilkan atau memampatkan, dalam hal ini data yang dimaksud adalah data komputer. Hal ini tentunya berkaitan erat dengan hal ekonomis, di mana bila kita dapat mengompresi data maka kita akan semakin sedikit memerlukan pembelian hardware untuk menyimpan data. Inti dari proses kompresi sendiri adalah menurunkan ukuran bit dari yang seharusnya menjadi kode yang memiliki ukuran lebih kecil.[5]

Berikut adalah beberapa contoh perhitungan sederhana dalam proses kompresi.

Contoh kebutuhan data selama 1 detik pada layar resolusi 640 x 480:

• Data text
   1. 1 karakter = 2 bytes (termasuk karakter ASCII Extended)
   2. Setiap karakter ditampilkan dalam 8x8 pixels
   3. Jumlah karakter yang dapat ditampilkan per halaman = (640 x 480) / (8 x 8) =4.800 karakter.

   Kebutuhan tempat penyimpanan per halaman = 4.800×2 byte = 9.600 byte = 9.375 Kbyte.

• Data Grafik Vektor
   1. 1 still image membutuhkan 500 baris
   2. Setiap 1 baris direpresentasikan dalam posisi horisontal, vertikal, dan field atribut sebesar 8-bit
   3. sumbu Horizontal direpresentasikan dengan log2 640 = 10 bits
   4. sumbu Vertical direpresentasikan dengan log2 480 = 9 bits
   5. Bits per line = 9bits + 10bits + 8bits = 27bits

   Kebutuhan tempat penyimpanan per halaman = 500 × (27 / 8) = 1687.5 byte = 1.65 Kbyte

• Color Display
   1. Jenis : 256, 4.096, 16.384, 65.536, 16.777.216 warna
   2. Masing-masing warna pixel memakan tempat 1 byte
   3. Misal 640 x 480 x 256 warna x 1 byte = 307.200 byte = 300 Kbyte[5]

Kompresi data sendiri dibagi dalam tiga klasifikasi dari bagaimana manusia (pengguna) akan menggunakan data itu selanjutnya. Tipe yang pertama adalah *Dialoque*, ini adalah tipe video conference yang tidak akan dibahas lebih lanjut, kemudian tipe *Losely* di mana data yang dikompresi akan digunakan sebagai data yang baru di mana sebagian data yang dianggap tidak penting dibuang. Tipe ini juga tidak akan terlalu banyak dibahas, contoh dari tipe *Losely* adalah resize gambar asli ke dalam gambar kecil untuk web. Sedang tipe terakhir yang akan banyak dibahas pada makalah ini adalah tipe *Loseless*. Tipe *Loseless* maksudnya data yang telah dikompresi tidak akan berubah karena akan digunakan sebagaimana data sebelum dikompresi. Contoh dari tipe ini merupakan ZIP, RAR, dan semacamnya. Kompresi data tipe yang terakhir tidak selalu menjadi lebih kecil ukurannya daripada sebelum dikompresi sebagai contoh .TAR akan membuat file menjadi lebih besar sebelum kemudian dikompresi kembali dengan .GZ sehingga data akan menjadi jauh lebih kecil.[4]

2.2.1. Algoritma Huffman Statis

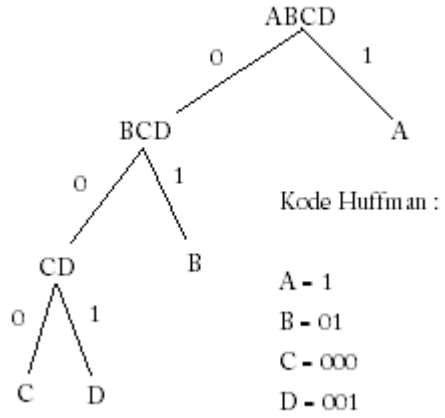
Representasi pembuatan Kode Huffman adalah representasi pohon prefiks. Pohon prefiks adalah pohon biner yang diakses dari daun terlebih dahulu mulai dari daun kiri, daun kanan kemudian akar. Pohon prefiks yang akar-akarnya diberi nama 0 dan 1 disebut kode prefix. Biasanya pada pohon prefiks daun kiri akan diberi nama 0 dan daun kanan akan diberi nama 1, kode biner tersebut inilah yang nantinya akan menjadi kode Huffman.

Untuk mendapatkan kode Huffman, mula-mula kita harus menghitung dulu kekerapan kemunculan tiap symbol di dalam teks. Cara pembentukan kode Huffman adalah dengan membentuk pohon biner, yang dinamakan pohon Huffman, sebagai berikut :

1. Pilih dua simbol dengan peluang paling kecil. Kedua simbol tadi dikombinasikan sebagai simpul orangtua sehingga menjadi simbol 2 karakter dengan peluang yang dijumlahkan
2. Selanjutnya pilih dua simbol berikutnya termasuk simbol baru yang mempunyai peluang terkecil.
3. Prosedur yang sama dilakukan pada dua simbol berikutnya yang mempunyai peluang terkecil.[2]

Contoh :
Kode ASCII *string* 7 huruf "ACABBDA" membutuhkan representasi 7 × 8 bit = 56 bit (7 byte), dengan rincian sebagai berikut:
A = 01000001
C = 01000010
A = 01000001
B = 01000011
B = 01000011
D = 01000100
A = 01000001
Pada *string* di atas, frekuensi kemunculan A = 3, B = 2, C = 1, dan D = 1.

Maka Pengerjaan akan seperti berikut :
-. C dan D menjadi 1 (CD) sehingga probabilitas menjadi 1/7 + 1/7 = 2/7
-. CD dan B menjadi 1 (BCD) sehingga probabilitas menjadi 2/7 + 2/7 = 4/7
-. BCD dan A menjadi 1 (ABCD) sehingga probabilitas menjadi 4/7 + 3/7 = 1
Perlu diingat bahwa probabilitas lebih besar akan diletakkan di kiri dengan pengkodean 0, sehingga pohon Huffman dari soal tersebut adalah. Alasan dari peletakkan ini akan dijelaskan pada bab berikutnya.



Gambar 1. Pohon Huffman untuk Karakter "ACABBDA"

Dengan melakukan pengkodean Huffman pada teks di atas kita dapat menghemat bit dari A yang sebelumnya 8 bit menjadi 1 bit, sehingga jumlah bit yang digunakan tadinya 8*4 bit = 32 bit menjadi 9 bit saja. Dapat dilihat jumlah bit setelah dilakukan pengkodean dengan algoritma Huffman dengan melihat table di bawah ini :

Huruf	Kode
A	1
B	01
C	000
D	001

Tabel 1. Kode Huffman untuk string "ACABBDA"

2.2.2. Proses Encoding

Encoding merupakan sebuah cara dalam menyusun susunan *(string)* biner dari data yang ada. Proses *encoding* pada pohon Huffman setelah kita memeroleh pohon Huffman seperti pada contoh di atas adalah dengan melakukan search infiks. Yaitu terlebih dahulu melihat akar baru kemudian akar kiri kemudian kanan. Karena inilah penyusunan pohon Huffman harus yang memiliki kemunculan lebih besar di kiri, karena pohon kiri akan diperiksa terlebih dahulu daripada pohon kanan. Pencarian dilakukan seterusnya sehingga akhirnya terbentuk suatu string biner baru yang merupakan hasil pengkodean Huffman dari data yang ada.
Langkah-langkah untuk men-*encoding* suatu string biner adalah sebagai berikut[3]

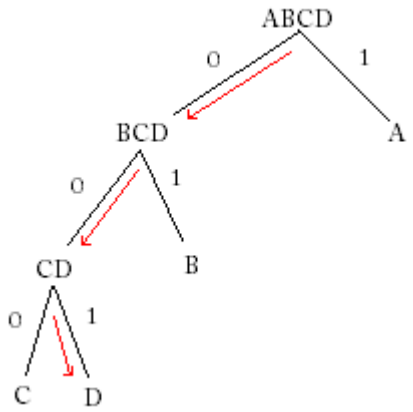
1. Tentukan karakter yang akan di-*encoding*
2. Mulai dari akar, baca setiap bit yang ada pada cabang yang bersesuaian sampai ketemu daun dimana karakter itu berada
3. Ulangi langkah 2 sampai seluruh karakter di*encoding*

2.2.3. Proses Decoding

Proses *Decoding* adalah kebalikan dari proses *encoding*. Arti dari *decoding* sendiri adalah menyusun kembali data yang telah dikodekan agar dapat dibaca dan dimanfaatkan. *Decoding* dapat dilakukan dengan dua cara, yang pertama dengan menggunakan pohon Huffman dan yang kedua dengan menggunakan tabel kode Huffman. Langkah-langkah men -*decoding* suatu *string* biner dengan menggunakan pohon Huffman adalah sebagai berikut :[3]

1. Baca sebuah bit dari *string* biner.
2. Mulai dari akar
3. Periksa kiri
4. Periksa kanan
5. Ulangi langkah 1, 2 dan 3 sampai bertemu daun. Kodekan rangkaian bit yang telah dibaca dengan karakter di daun.
6. Ulangi dari langkah 1 sampai semua bit di dalam *string* habis.

Sebagai contoh kita akan men-decoding string biner yang bernilai ”001”



Gambar 2. Proses Decoding dengan Menggunakan Pohon Huffman

setelah kita telusuri dari akar, maka kita akan menemukan bahwa string yang mempunyai kode Huffman “001” adalah karakter D. Cara yang kedua adalah dengan menggunakan tabel kode Huffman. Sebagai contoh kita akan menggunakan kode Huffman pada Tabel 1 untuk merepresentasikan *string* “ACABBDA”. Dengan menggunakan Tabel 1 *string* tersebut akan direpresentasikan menjadi rangkaian bit: 1 000 1 01 01 001 1. Jadi, jumlah bit yang dibutuhkan hanya 13 bit.

Dari Tabel 1 tampak bahwa kode untuk sebuah simbol/karakter tidak boleh menjadi awalan dari kode simbol yang lain guna menghindari keraguan (ambiguitas) dalam proses dekompresi atau *decoding*. Karena tiap kode Huffman yang dihasilkan unik, maka proses *decoding* dapat dilakukan dengan mudah. Contoh: saat membaca kode bit pertama dalam rangkaian bit “1000101010011”, yaitu bit “1”, dapat langsung disimpulkan bahwa kode bit “1” merupakan pemetaan dari simbol “A”. Kemudian baca kode bit selanjutnya, yaitu bit “0”. Tidak ada kode Huffman “0”, lalu baca kode bit selanjutnya, sehingga menjadi “00”. Tidak ada juga kode Huffman “00”, lalu baca lagi kode bit berikutnya, sehingga menjadi “000”. Rangkaian kode bit “000” adalah pemetaan dari simbol “C” sehingga karakter kedua adalah “C”, berikutnya kembali 1 langsung ditemukan sehingga karakter “A”, lalu “01” berulang 2 kali menandakan “BB” dan terakhir “011” adalah “D”, sehingga dapat diperoleh kembali bahwa string input adalah “ACABBDA”

2.3. Algoritma Huffman Dinamis

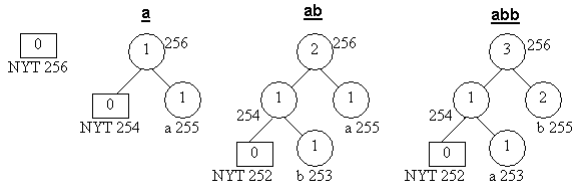
Seperti telah dikatakan pada pendahuluan, bahwa akan dijelaskan algoritma lain selain Algoritma Huffman (statis) atau SHC (Static Huffman Coding) yang telah dipelajari pada kuliah Struktur Distkrit. Ialah Adaptive Huffman Coding (AHC) atau Algoritma Huffman Dinamis, sebuah kelanjutan dari algoritma Huffman Statis (seperti telah dikatakan juga bahwa algoritma Huffman adalah dasar dari kompresi data). AHC seperti SHC juga merupakan kompresi *loseless* namun lebih efisien sehingga data yang terbentuk akan lebih cepat dengan ukuran yang sama. Namun AHC akan memerlukan sedikit lebih banyak memori sementara karena harus menyimpan nilai maksimal, oleh karena itu AHC lebih sering dipakai dalam proses transfer data (streaming) dan SHC dipakai dalam proses pengompresian internal.

Berikut algoritma AHC dan mengapa AHC dapat membuat kompresi yang lebih kecil daripada SHC

Algoritma AHC FGK (Faller-Gallager-Knuth):[1]

1. Buat suatu simbol yang bernama NYT (Not Yet Transmitted) atau berarti belum ditransmisi.
2. Jika symbol adalah NYT, tambahkan 2 anak NYT tersebut, satunya akan menjadi NYT baru dan yang lainnya adalah untuk symbol, akan menambahkan nilai dari NYT akar dan daun yang terbentuk, jika daun baru terbentuk lanjutkan ke langkah 4, jika telah ada sebelumnya periksa dulu di langkah 3.
3. Jika symbol yang dimasukkan terakhir memiliki probabilitas yang lebih tinggi maka akan ditukar dengan yang sebelumnya menempati tempat kemunculan tertinggi.
4. Tambahkan nilai dari akar tersebut.
5. Jika bukan akar, kembali ke akar inang lalu lanjutkan ke langkah 2, selain itu selesai.

Contoh :



Gambar 3. Pembentukan pohon kode algoritma Huffman Dinamis

1. Pohon Huffman kosong dan berisi NYT256(2byte)
2. Karakter pertama adalah a, a disimpan pada anak NYT 256 yang bobotnya bertambah 1 dan menambah anak NYT 254 dan a berbobot 1 dengan alamat 255
3. Karakter berikutnya ialah b, b disimpan pada anak NYT254, NYT 254 bertambah bobotnya menjadi 1 dan b dengan alamat 253 dan terbentuk NYT baru 252.
4. Karakter berikutnya adalah b lagi maka b akan dimasukkan ke 253 (searching) dank arena nilainya bertambah akan dibandingkan dengan alamat maksimum yakni 255 dan ternyata bobot b lebih besar sehingga terjadi pertukaran nilai antara a dan b

Sekilas AHC dan SHC terlihat sama, namun justru yang menarik dari algoritma AHC adalah pembuatan peta kode yang berubah-ubah atau seringkali dikatakan peta kode yang "belajar".

Pada langkah ke-2 a memiliki kode "1" dan b memiliki kode "01" namun pada langkah ke-3 a menjadi kode "01" dan b "1". Inilah yang membuat algoritma AHC lebih efisien dan mampat.[1]

Proses encoding dan decoding dari Adaptive Huffman Coding sama dengan proses encoding dan decoding dari Static Huffman Coding namun dengan peta kode yang terus berubah.

3. HASIL DAN PEMBAHASAN

Pseudocode pembuatan pohon kode Algoritma Huffman:

Procedure Huffman (**Input**: list of n: W, integer: n, **Output**: Pohon biner berbobot: T)
Createlistpohon F dari elemen W {membuat pohon dari list W}
While (IsDaun(F) = false) do
    T1 ← min(F) {min mengambil data minimum F}
    T2 ← min(F)
    Tree (T, T1, T2) {prosedur Tree membuat pohon biner T dengan daun T1 dan T2, insert first sebagai akarnya adalah hasil jumlah T1 dan T2)

Sedangkan pseudocode Static Huffman Code

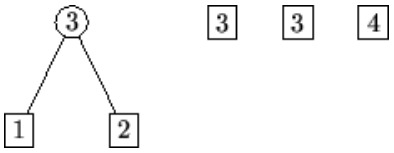
Huffman(W,n,T)

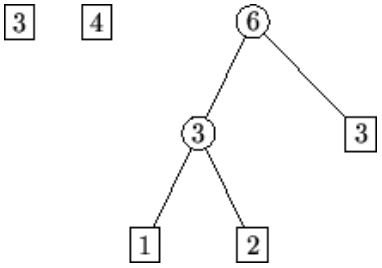
Procedure SHC (**Input**: pohon biner: T, **Output**: Data terkompres: D)
D ← hasil Decoding char c dari pohon biner T

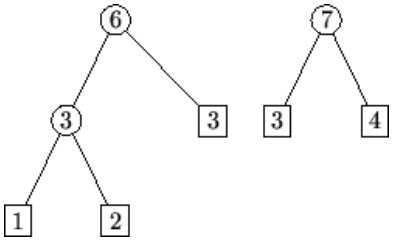
Dengan Huffman selama log n dan SHC sebanyak n kali. Sehingga rumus waktu asimptotiknya *O(n log n)*.

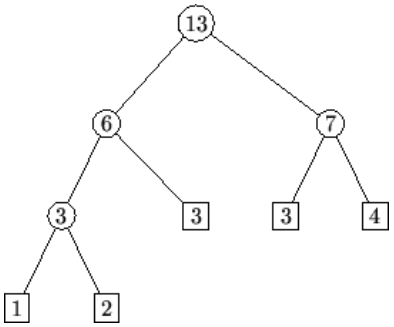
Maka jika kita memiliki data "eddbcaeecdceb" {a,b,c,d,e} dengan bobot {1,2,3,3,4} akan terjadi proses seperti berikut:

1. 

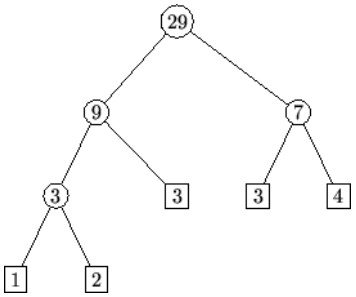
2. 

3. 

4. 

5. 

6.



Gambar 4. Urutan pembuatan pohon dari contoh secara statis

Sehingga bila yang ingin dikode adalah "eddbcaeecdceb" yang terkode adalah "11 10 10 001 01 000 11 11 01 10 01 11 10" atau 28 bit.

Tiap iterasi algoritma Huffman statis mereduksi permasalahan sebanyak 1 sehingga akan terjadi n buah iterasi. Iterasi ke-i teridir dari menempatkan 2 nilai minimum pada iterasi ke *n-i+1* pada list. Karena itu kompleksitas algoritma Huffman adalah operasi linear dank arena dilakukan 2 kali berulang maka memiliki kompleksitas waktu asimptotik $O(n^2)$.

Bagaimanapun, ide baru yaitu mencari kekerapan pada awal pembentukan list dengan membentuk sebuah list yang berisi kekerapan masing-masing elemen. Inisialisasi ini mereduksi kompleksitas waktu asimptotik algoritma menjadi *O(n log n)*. Proses mencari dua buah nilai minimum pohon dengan mengambil dari pohon pertama kemudian membandingkannya dengan pohon kekerapan dari yang terbesar mempercepat algoritma. Karena saat memasukkan elemen ke-i pembandingan dengan elemen list kekerapan dilakukan dengan i-1 elemen yang sebelumnya telah dimasukkan maka algoritma ini akan mengganti iterasi sebanyak n kali yang dilakukan pada algoritma sebelumnya, hal ini yang menyebabkan algoritma Huffman memiliki kompleksita *O(n log n)*.

Banyaknya simpul pada pohon algoritma Huffman adalah 2n-1 karena tiap proses membentuk 2 buah daun dengan proses sebanyak n dan proses terakhir hanya membentuk 1 daun.

Pseudocode Algoritma Huffman Dinamis merupakan kelanjutan dari algoritma Huffman biasa di mana pembuatan pohon dilakukan berkali-kali dengan input beragam:

```
Procedure AHC (Input: list of n: W, string: s, char: c,
Output: Data terkompres: D)
integer: n
pohon biner: T
n ← banyak karakter c di s
Huffman(W,n,T)
D ← hasil Decoding char c dari pohon biner T
```

Dengan kompleksitas waktu asimptotik Huffman dan decoding adalah log n sehingga AHC memiliki kompleksitas waktu asimptotik *O(log n)*.

Untuk kasus yang sama akan terjadi perubahan peta kode:
"eddbcaeecdce"

1. "e"

Data	Kekerapan	Kode
e	1	1

2. "ed"

Data	Kekerapan	Kode
e	1	1
d	1	01

3. "edd"

Data	Kekerapan	Kode
e	1	01
d	2	1

4. "eddb"

Data	Kekerapan	Kode
e	1	01
d	2	1
b	1	00

5. "eddbc"

Data	Kekerapan	Kode
e	1	01
d	2	1
b	1	001
c	1	000

6. "eddbca"

Data	Kekerapan	Kode
e	1	011
d	2	1
b	1	010
c	1	000
a	1	001

7. "eddbcae"

Data	Kekerapan	Kode
e	2	11
d	2	01
b	1	100
c	1	101
a	1	00

8. "eddbcaee"

Data	Kekerapan	Kode
e	3	11
d	2	01
b	1	100
c	1	101
a	1	00

9. "eddbcaeec"

Data	Kekerapan	Kode
e	3	11
d	2	01
b	1	101
c	2	00
a	1	100

10. “eddbcaeecd”

Data	Kekerapan	Kode
e	3	11
d	3	10
b	1	001
c	2	01
a	1	000

11. “eddbcaeecdc”

Data	Kekerapan	Kode
e	3	11
d	3	10
b	1	001
c	3	01
a	1	000

12. “eddbcaeecdce”

Data	Kekerapan	Kode
e	4	11
d	3	10
b	1	001
c	3	01
a	1	000

13. “eddbcaeecdceb”

Data	Kekerapan	Kode
e	4	11
d	3	10
b	2	001
c	3	01
a	1	000

Tabel 2. Urutan perubahan tabel kode Algoritma Huffman dinamik untuk soal contoh

Tiap kali terjadi input, pohon kode akan diubah seperti membuat pohon baru dengan data yang kini didapatkan dan output langsung dikodekan, pembuatan pohon i elemen akan memerlukan waktu asimptotik algoritma sebanyak log n atau proses AHC memiliki O(*log n*) yang lebih cepat daripada proses SHC (*O(n log n)*).

Sehingga yang terkode adalah “1 01 1 00 000 001 11 11 00 10 01 11 001” atau hanya 27 bit kasus ini hanya berbeda sedikit karena memang datanya hanya sedikit, namun coba bayangkan dalam transmisi data yang berukuran sangat besar dan perbedaan macam data yang banyak.

Setelah melihat perbandingan dan bagaimana algoritma Huffman dapat mengompresi data kita dapat merasakan manfaatnya sekarang ini di mana data tidak lagi berukuran seperti perhitungan pada bagian pendahuluan melainkan dapat dibuat lebih kecil karena data saat disimpan dalam harddisk dikompres terlebih dahulu dan didekompresi saat kita membukanya, walaupun beberapa kompresi tersebut adalah bukan dengan algoritma Huffman dan tipe *loseless* yang dibahas di sini namun algoritma Huffman dan kompresi *loseless* adalah cikal bakal kompresi data.

4. KESIMPULAN

Algoritma Huffman terbukti ampuh dalam melakukan kompresi data, walaupun algoritma Huffman sudah sangat lama diciptakan namun pengaruhnya hingga kini pada dunia pengompresian data sangatlah besar. Walaupun banyak algoritma lain telah ditemukan untuk mengompresi data, Algoritma Huffman Statik masih digunakan dan Algoritma Huffman Dinamis merupakan algoritma kompresi yang penting dalam kompresi data internal maupun streaming yang bersifat loseless.

Algoritma Huffman dinamis memang lebih efektif dan efisien jika dibandingkan dengan Algoritma Huffman statis tapi itu bukan berarti Algoritma Huffman statis buruk sama sekali karena dari Algoritma Huffman statis ini terbentuk berbagai macam algoritma dekompresi lanjut. Algoritma Huffman dinamis merupakan salah satunya yang memiliki waktu kompresi sangat cepat namun hasilnya belum terlalu mampat (walaupun lebih mampat daripada Algoritma Huffman statis).

Karena pentingnya kompresi data dalam dunia modern ini maka pembelajaran dan penelitian mengenai kompresi data hendaknya terus dilanjutkan, sekian yang dapat saya katakan sebagai mahasiswa yang baru mulai mempelajari pengompresian data komputer.

DAFTAR REFERENSI

[1] Leweler, A. Debra and Daniel S. Hirchsberg, *“Data Compression”*, journal.

[2] Munir, Rinaldi, *Struktur Diskrit*, Program Studi Teknik Informatika STEI ITB, 2008

[3] Wardoyo, Irwan dkk. “Kompresi Teks dengan menggunakan Algoritma Huffman”, journal

[4] *http://www.ics.uci.edu/~dan/pubs/DC-Sec4.html*, 30 Desember 2008

[5] *http://one.indoskripsi.com/judul-skripsi-tugas-makalah/teknologi-informasi-lanjutan/kompresi-data-dengan-algoritma-huffman-dan-algoritma-lainnya,* 31 Desember 2008