PERBANDINGAN RASIO DAN KECEPATAN KOMPRESI MENGGUNAKAN ALGORITMA HUFFMAN, LZW DAN DMC

Nancy J.Tuturoong

ABSTRAK

Kompresi ialah proses pengubahan sekumpulan data menjadi suatu bentuk kode untuk menghemat kebutuhan tempat penyimpanan dan waktu untuk transmisi data.
Dengan menggunakan algoritma Huffman, proses pengompresan teks dilakukan dengan menggunakan prinsip pengkodean, yaitu tiap karakter dikodekan dengan rangkaian beberapa bit sehingga menghasilkan hasil yang lebih optimal.
LZW adalah repeated-string compressor, LZW menggunakan kamus data (atau yang sering disebut dengan translation table atau string table) untuk merepresentasikan agar data menjadi linier di dalam uncompressed input stream.
DMC merupakan teknik kompresi yang adaptif, karena struktur mesin finite-state berubah seiring dengan pemrosesan file. Kemampuan kompresinya tergolong amat baik, meskipun waktu komputasi yang dibutuhkan lebih besar dibandingkan metode lain.
Secara rata-rata algoritma Huffman membutuhkan waktu kompresi yang tersingkat, diikuti algoritma LZW dan yang terakhir algoritma DMC. LZW mengorbankan kecepatan kompresi untuk mendapatkan rasio hasil kompresi yang baik.

Kata Kunci : kompresi, huffman, LZW, DMC

I. PENDAHULUAN

I.1 Latar Belakang Masalah

Metode - metode kompresi data sangat dibutuhkan untuk mengkompresi data yang memiliki kapasitas ukuran besar ke dalam ukuran yang kecil untuk menghemat penggunaan memori. Contohnya teks yang merupakan kumpulan dari karakter – karakter atau *string* yang menjadi satu kesatuan. Teks yang memuat banyak karakter didalamnya selalu menimbulkan masalah pada media penyimpanan dan kecepatan waktu pada saat transmisi data. Media penyimpanan yang terbatas, membuat semua orang mencoba berpikir untuk menemukan sebuah cara yang dapat digunakan untuk mengkompres teks.

Algoritma Huffman, yang dibuat oleh seorang mahasiswa MIT bernama David Huffman, merupakan salah satu metode paling lama dan paling terkenal dalam kompresi teks. Algoritma Huffman menggunakan prinsip pengkodean yang mirip dengan kode Morse, yaitu tiap karakter (simbol) dikodekan hanya dengan rangkaian beberapa bit, dimana karakter yang sering muncul dikodekan dengan rangkaian bit yang pendek dan karakter yang jarang muncul dikodekan dengan rangkaian bit yang lebih panjang.

LZW adalah algoritma kompresi lossless, antara kompresi dan dekompresi waktunya adalah simetris. Pertama kali suatu urutan ditemukan kode yang berbeda maka kode tersebut dan ditambahkan kedalam kamus data. Semua data yang ada dibandingkan dengan data masukan, jika sama maka diwakilkan dengan sebuah kode.

Algoritma DMC merupakan teknik pemodelan yang didasarkan pada model *finite-state* (model Markov). Kemampuan kompresinya tergolong amat baik, meskipun waktu komputasi yang dibutuhkan lebih besar dibandingkan metode lain.

Metode kompresi yang pertama kali muncul bisa dikatakan sebagai perintis metode kompresi data. LZ77 dan variannya (LZ78, LZW, GZIP), *Dynamic Markov Compression* (DMC), *Block-Sorting Lossless*, *Run-Length*, *Shannon-Fano*, *Arithmetic*, PPM (*Prediction by Partial Matching*), *Burrows-Wheeler Block Sorting*, dan *Half Byte*.

Ada beberapa faktor yang sering menjadi pertimbangan dalam memilih suatu metode kompresi yang tepat, yaitu kecepatan kompresi, sumber daya yang dibutuhkan (memori, kecepatan PC), ukuran file hasil kompresi, besarnya redundansi, dan kompleksitas algoritma.

I.2 Perumusan Masalah

Tinjauan yang akan dibahas adalah :

- ✓ Metode – metode kompresi data khususnya algoritma Huffman, algoritma LZW dan algoritma DMC
- ✓ Algoritma kompresi data khususnya algoritma Huffman, algoritma LZW dan algoritma DMC.
- ✓ Perbandingan metode – metode kompresi data khususnya algoritma Huffman, algoritma LZW dan algoritma DMC.

I.3 Batasan Masalah

Hanya memaparkan 3 buah metode kompresi data yaitu :

- Algoritma Huffman,
- Algoritma LZW dan
- Algoritma DMC.

I.4 Tujuan

Berdasarkan latar belakang masalah yang ada di atas maka pembahasan tugas ini bertujuan untuk mengetahui metode – metode kompresi data, algoritma, cara – cara mengkompresi data serta perbandingan metode – metode kompresi data, khususnya algoritma Huffman, algoritma LZW dan algoritma DMC.

I.5 Manfaat

Penulisan ini diharapkan dapat bermanfaat :

1. Sebagai referensi dalam memahami metode – metode kompresi data dalam hal ini ada 3 yaitu : algoritma Huffman, algoritma LZW dan algoritma DMC.

2. Sebagai bahan untuk mempelajari lebih lanjut metode – metode kompresi data, khususnya algoritma Huffman, algoritma LZW dan algoritma DMC.

II.TINJAUAN PUSTAKA

2.1 Algoritma Huffman

Berdasarkan tipe peta kode yang digunakan untuk mengubah pesan awal (isi data yang diinputkan) menjadi sekumpulan *codeword*, algoritma Huffman termasuk kedalam kelas algoritma yang menggunakan metode statik . Metoda statik adalah metoda yang selalu menggunakan peta kode yang sama, metoda ini membutuhkan dua fase (*two-pass*): fase pertama untuk menghitung probabilitas kemunculan tiap simbol dan menentukan peta kodenya, dan fase kedua untuk mengubah pesan menjadi kumpulan kode yang akan di taransmisikan.

Sedangkan berdasarkan teknik pengkodean simbol yang digunakan, algoritma Huffman menggunakan metode *symbolwise*. Metode *symbolwise* adalah metode yang menghitung peluang kemunculan dari setiap simbol dalam satu waktu, dimana simbol yang lebih sering muncul diberi kode lebih pendek dibandingkan simbol yang jarang muncul.

2.1.1 Pembentukan Pohon Huffman

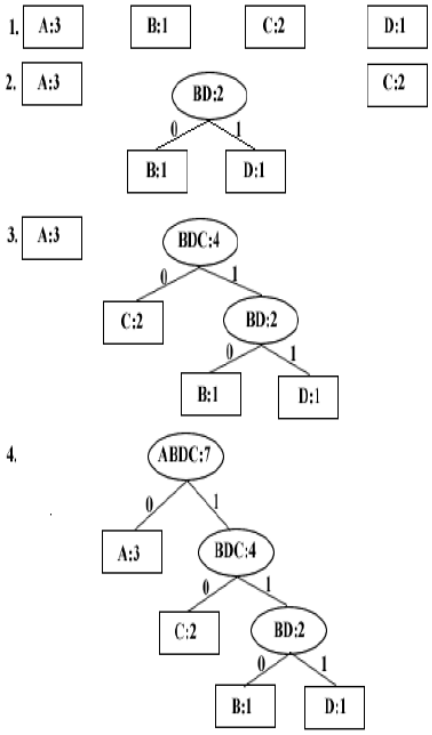
Kode Huffman pada dasarnya merupakan kode prefiks (*prefix code*). Kode prefiks adalah himpunan yang berisi sekumpulan kode biner, dimana pada kode prefik ini tidak ada kode biner yang menjadi awal bagi kode biner yang lain. Kode prefiks biasanya direpresentasikan sebagai pohon biner yang diberikan nilai atau label. Untuk cabang kiri pada pohon biner diberi label 0, sedangkan pada cabang kanan pada pohon biner diberi label 1. Rangkaian bit yang terbentuk pada setiap lintasan dari akar ke daun merupakan kode prefiks untuk karakter yang berpadanan. Pohon biner ini biasa disebut pohon Huffman.

Sebagai contoh, dalam kode ASCII *string* 7 huruf “ABACCDA” membutuhkan representasi

7 × 8 bit = 56 bit (7 byte), dengan rincian sebagai berikut:

A = 01000001
B = 01000010
A = 01000001
C = 01000011
C = 01000011
D = 01000100
A = 01000001

Pada *string* di atas, frekuensi kemunculan A = 3, B = 1, C = 2, dan D = 1,



Gambar 1. Pohon Huffman untuk Karakter "ABACCDA"

2.1.2 Kompleksitas Algoritma Huffman

Algoritma Huffman mempunyai kompleksitas waktu O(n log n), karena dalam melakukan sekali proses itersi pada saat penggabungan dua buah pohon yang mempunyai frekuensi terkecil pada sebuah akar membutuhkan waktu O(log n), dan proses itu dilakukan berkali-kali sampai hanya tersisa satu buah pohon Huffman itu berarti dilakukan sebanyak n kali[4].

2.2 Algoritma Greedy

Algoritma greedy adalah algoritma yang mecahkan masalah langkah per langkah, pada setiap langkahnya algoritma greedy melakukan :

1. Mengambil pilihan yang terbaik yang dapat diperoleh pada saat itu tanpa memperhatikan konsekuensi ke depan (prinsip "*take what you can get now!*")

2. Berharap bahwa dengan memilih optimum lokal pada setiap langkah akan berakhir dengan optimum global.

2.2.1 Hubungan Algoritma Greedy dengan Algoritma Huffman

Pada awalnya David Huffman hanya men*encoding* karakter dengan hanya menggunakan pohon biner biasa, namun setelah itu David Huffman menemukan bahwa penggunaan algoritma *greedy* dapat membentuk kode prefiks yang optimal.

Penggunaan algoritma greedy pada algoritma Huffman adalah pada saat pemilihan dua pohon dengan frekuensi terkecil dalam membuat pohon Huffman. Oleh karena itu algoritma Huffman adalah salah satu contoh algoritma yang menggunakan dari algoritma greddy.

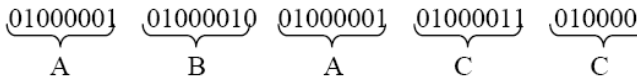
Sebagai contoh terdapat sebuah teks yang terdiri dari 120 karakter, yang masing-masing karakter mempunyai *cost*. Tujuan kita adalah menghitung total *cost* yang dikeluarkan untuk membentuk teks tersebut.

Karakter	cost	Kode Huffman	Total cost
A	10	000	10x3=30
B	15	010	15x3=45
C	5	0010	5x5=25
D	15	011	15x3=45
E	20	111	20x3=60
F	5	00110	5x5=25
G	15	110	15x3=45
H	30	10	30x2=60
I	5	00111	5x5=25
Total cost			360

Tabel 1. Contoh perhitungan total cost pada suatu teks

Sebagai contoh, dalam kode ASCII *string* 7 huruf "ABACCDA" membutuhkan representasi 7 × 8

bit = 56 bit (7 byte), dengan rincian sebagai berikut:



Untuk mengurangi jumlah bit yang dibutuhkan, panjang kode untuk tiap karakter dapat dipersingkat, terutama untuk karakter yang frekuensi kemunculannya besar. Pada *string* di atas, frekuensi kemunculan A = 3, B = 1, C = 2, dan D = 1, sehingga dengan menggunakan algoritma di atas diperoleh kode Huffman seperti pada Tabel 2.

Tabel 2. Kode Huffman utnuk ”ABACCSA”

Simbol	Frekuensi	Peluang	Kode Huffman
A	3	3/7	0
B	1	1/7	110
C	2	2/7	10
D	1	1/7	111

Dengan menggunakan kode Huffman ini, *string* “ABACCDA” direpresentasikan menjadi rangkaian bit : 0 110 0 10 10 111 0. 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 dekompresi dapat dilakukan dengan mudah.

Algoritma Huffman menghasilkan kompresi data dengan cara mengkodekan data berdasarkan frekuensi kemunculan nilainya. Struktur data yang digunakan untuk mengkodekan data adalah suatu *weighted binary tree*, atau berupa suatu *Huffman tree* (pohon Huffman).

Pengkodean Huffman digunakan untuk kompresi data yang bersifat *lossless*. Pada pengkodean Huffman, nilai yang sering muncul dikodekan dalam jumlah bit yang lebih pendek, sedangkan nilai yang jarang muncul dikodekan dalam jumlah bit yang lebih panjang. Pengkodean ini menghasilkan *variable-length codes* dengan redundansi minimal

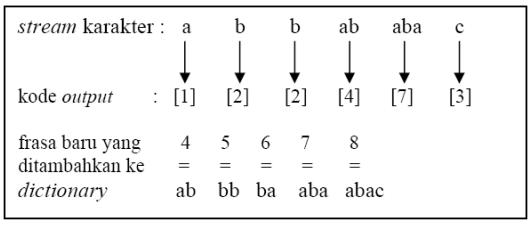
2..3. Algoritma LZW *(Lempel-Ziv-Welch)*

Algoritma ini melakukan kompresi dengan menggunakan *dictionary*, di mana fragmen-fragmen teks digantikan dengan indeks yang diperoleh dari sebuah “kamus”. Pendekatan ini bersifat *adaptif* dan efektif karena banyak karakter dapat dikodekan dengan mengacu pada *string* yang telah muncul sebelumnya dalam teks. Prinsip kompresi tercapai jika referensi dalam bentuk *pointer* dapat disimpan dalam jumlah bit yang lebih sedikit dibandingkan *string* aslinya.

Sebagai contoh, string “ABBABABAC” akan dikompresi dengan LZW. Isi *dictionary* pada awal proses diset dengan tiga karakter dasar yang ada: “A”, “B”, dan “C”. Kolom *dictionary* menyatakan *string* baru yang sudah ditambahkan ke dalam *dictionary* dan nomor indeks untuk *string* tersebut ditulis dalam kurung siku. Kolom *output* menyatakan kode output yang dihasilkan oleh langkah kompresi. Hasil proses kompresi ditunjukkan pada Tabel 3.

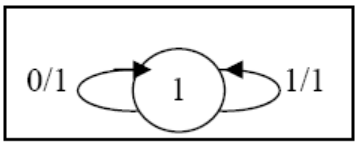
Tabel 3. Tahapan proses kompresi LZW

Langkah	Posisi	Karakter	*Dictionary*	Output
1.	1	A	[4] A B	[1]
2.	2	B	[5] B B	[2]
3.	3	B	[6] B A	[2]
4.	4	A	[7] A B A	[4]
5.	6	A	[8] A B A C	[7]
6.	9	C	- - -	[3]



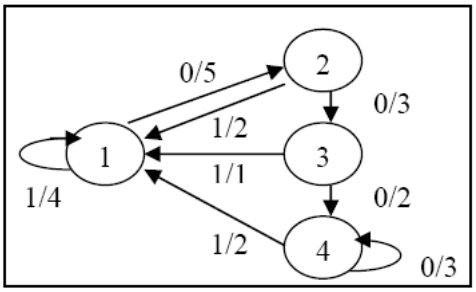
2.4. Algoritma DMC *(Dynamic Markov Compression)*

Algoritma DMC merupakan teknik pemodelan yang didasarkan pada model *finite-state* (model Markov). DMC membuat probabilitas dari karakter biner berikutnya dengan menggunakan pemodelan *finite-state*, dengan model awal berupa mesin FSA dengan transisi 0/1 dan 1/1, seperti ditunjukkan pada Gambar 2. DMC merupakan teknik kompresi yang adaptif, karena struktur mesin *finite-state* berubah seiring dengan pemrosesan file. Kemampuan kompresinya tergolong amat baik, meskipun waktu komputasi yang dibutuhkan lebih besar dibandingkan metode lain.



Gbr 2. Model awal DMC

Pada DMC, simbol alfabet input diproses per bit, bukan per byte. Setiap output transisi menandakan berapa banyak simbol tersebut muncul. Penghitungan tersebut dipakai untuk memperkirakan probabilitas dari transisi. Contoh: pada Gambar 3, transisi yang keluar dari *state* 1 diberi label 0/5, artinya bit 0 di state 1 terjadi sebanyak 5 kali.

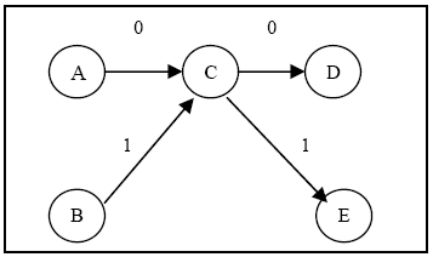


Gbr 3. Sebuah model yang diciptakan oleh metode DMC

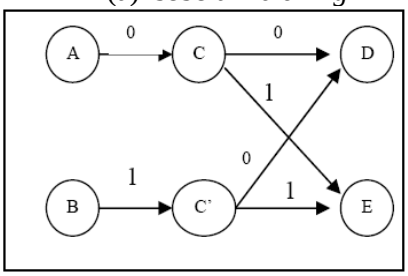
Secara umum, transisi ditandai dengan 0/*p* atau 1/*q* dimana *p* dan *q* menunjukkan jumlah transisi dari *state* dengan input 0 atau 1. Nilai probabilitas bahwa input selanjutnya bernilai 0 adalah *p*/(*p*+*q*) dan input selanjutnya bernilai 1 adalah *q*/(*p*+*q*). Lalu bila bit sesudahnya ternyata bernilai 0, jumlah bit 0 di transisi sekarang ditambah satu menjadi *p*+1. Begitu pula bila bit sesudahnya ternyata bernilai 1, jumlah bit 1 di transisi sekarang ditambah satu menjadi *q*+1.

Jika frekuensi transisi dari suatu *state t* ke *state* sebelumnya, yaitu *state u*, sangat tinggi, maka *state t* dapat di-*cloning*. Ambang batas nilai *cloning* harus disetujui oleh *encoder* dan *decoder*. State yang di-*cloning* diberi simbol *t*'. Aturan *cloning* adalah sebagai berikut :

- Semua transisi dari *state u* dikirim ke *state t*'. Semua transisi dari *state* lain ke *state* t tidak berubah.
- Jumlah transisi yang keluar dari *t*' harus mempunyai rasio yang sama (antara 0 dan 1) dengan jumlah transisi yang keluar dari *t*.
- Jumlah transisi yang keluar dari *t* dan *t*' diatur supaya mempunyai nilai yang sama dengan jumlah transisi yang masuk.



(a) sebelum *cloning*



(b) setelah *cloning*

Gambar 4. Model Markov sebelum dan setelah *cloning*

Metode DMC yang diterapkan dalam penelitian ini bertipe dinamik, dimana hanya dilakukan satu kali pembacaan terhadap file input. Kalkulasi dilakukan secara *on the fly* (proses perhitungan probabilitas dilakukan bersamaan dengan pengkodean data).

III. METODOLOGI PENELITIAN

3.1 Tempat dan Waktu Penelitian

Pelaksanaan penelitian dilakukan di Fakultas Teknik Universitas Sam Ratulangi Manado, laboratorium komputer dengan waktu 3 bulan.

3.2 Bahan dan Alat yang digunakan

1. Spesifikasi komputer yang digunakan :
   Komputer 1 :
   Computer name : USER-46842A0644
   Operating System : Microsoft Windows XP Professional (5.1, Build 2600)

Language : English (Regional Setting : English)
System Manufacture : Phoenix/SiS
System Model : M720S
BIOS : BIOS Revision: 1.00.03
Pro essor : Genuine Intel(R) CPU T2130 @ 1.86GHz (2 CPUs)
Memory : 382MB RAM
Page file : 400MB used, 517MB available
DirectX Version : 9.0c (4.09.0000.0904)

Komputer 2 :

Computer name : LSK-UNKNOWN
Operating System : Microsoft Windows XP Professional (5.1, Build 2600)
Language : English (Regional Setting : English)
System Manufacturer : System Manufacturer
System Model : System Product Name
BIOS : BIOS Date : 07/12/06 00:28:03 Ver: 08.00.10
Processor : Intel(R) Pentium(R) D CPU 3.40GHz (2 CPUs)
Memory : 512MB RAM
Page file : 366MB used, 882MB available
DirectX Version : DirectX 9.0c (4.09.0000.0904)

2. Software (Huffman Compressor, LZW compressor dan DMC)
3. Net-working digunakan untuk mencari data yang dibutuhkan.
4. File yang akan diuji
5. Flash Drive dan CD sebagai media penyimpan data.

3.3 Prosedur Penelitian

Prosedur yang dilakukan yaitu :

1. Studi literatur, yakni mencari dan mempelajari metode – metode pengkompresan data.
2. Pengujian adalah pengujian metode kompresi data tersebut dengan file – file yang sudah disiapkan terlebih dahulu kemudian menginstal software yang akan digunakan dalam.
3. Analisa dengan membandingkan data yang di dapat dari pengujian pengkompresian data yang telah diuji.

IV. PEMBAHASAN

4. 1 Algoritma Huffman

4.1.1 Proses Encoding

Encoding adalah cara menyusun *string* biner dari teks yang ada. Proses *encoding* untuk satu karakter dimulai dengan membuat pohon Huffman terlebih dahulu. Setelah itu, kode untuk satu karakter dibuat dengan menyusun nama *string* biner yang dibaca dari akar sampai ke daun pohon Huffman.

Langkah-langkah untuk men-*encoding* suatu string biner adalah sebagai berikut

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 kara kter di*encoding* Sebagai contoh kita dapat melihat tabel dibawah ini, yang merupakan hasil *encoding* untuk pohon Huffman pada gambar 1

Karakter	String Biner Huffman
A	0
B	110
C	10
D	111

Table 5. Kode Huffman untuk Karakter “ABCD”

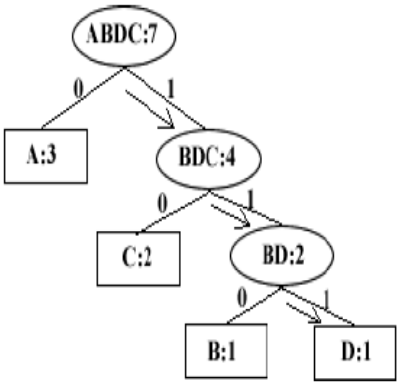
4.1.2 Proses Decoding

Decoding merupakan kebalikan dari *encoding*. *Decoding* berarti menyusun kembali data dari *string* biner menjadi sebuah karakter kembali. *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 :

1. Baca sebuah bit dari *string* biner.
2. Mulai dari akar
3. Untuk setiap bit pada langkah 1, lakukan traversal pada cabang yang bersesuaian.

4. Ulangi langkah 1, 2 dan 3 sampai bertemu daun. Kodekan rangkaian bit yang telah dibaca dengan karakter di daun.
5. Ulangi dari langkah 1 sampai semua bit di dalam *string* habis. Sebagai contoh kita akan men-decoding string biner yang bernilai ”111”



Gbr 5. Proses Decoding dengan Menggunakan Pohon Huffman

Setelah kita telusuri dari akar, maka kita akan menemukan bahwa string yang mempunyai kode Huffman “111” adalah karakter D.

Cara yang kedua adalah dengan menggunakan tabel kode Huffman. Sebagai contoh kita akan menggunakan kode Huffman pada Tabel 5 untuk merepresentasikan *string* “ABACCDA”. Dengan menggunakan Tabel 1 *string* tersebut akan direpresentasikan menjadi rangkaian bit : 0 110 0 10 10 1110. Jadi, jumlah bit yang dibutuhkan hanya 13 bit. Dari Tabel 5 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 “011001010110”, yaitu bit “0”, dapat langsung disimpulkan bahwa kode bit “0” merupakan pemetaan dari simbol “A”. Kemudian baca kode bit selanjutnya, yaitu bit “1”. Tidak ada kode Huffman “1”, lalu baca kode bit selanjutnya, sehingga menjadi “11”. Tidak ada juga kode Huffman “11”, lalu baca lagi kode bit berikutnya, sehingga menjadi “110”. Rangkaian kode bit “110” adalah pemetaan dari simbol “B”.

4.1.3 Pengujian Algoritma Huffman

Pada pengujian digunakan, kita akan men*encoding* sebuah teks yang berisi 100.000 string, diantaranya 45.000 karakter ‘g’, 13.000 karakter ‘o’, 12.000 karakter ‘p’, 16.000 karakter ‘h’, 9.000 karakter ‘e’, dan 5.000 karakter ‘r’ dengan menggunakan 3 cara, yaitu dengan menggunakan kode ASCII , kode 3-bit dan kode Huffman. Setelah itu ketiga kode tersebut akan dibandingkan satu sama lainnya.

Kode ASCII

Karakter	ASCII	Biner
g	103	1100111
o	111	1101111
p	112	1110000
h	104	1101000
e	101	1100101
r	114	1110010

Tabel 6. Kode ASCII untuk karakter “ g,o,p,h,e,r,”

Untuk meng-encoding teks tersebut kita membutuhkan sebanyak

untuk karakter ‘g’ 45.000 x 8 bit (1100111) = 360.000 bit
untuk karakter ‘o’ 13.000 x 8bit (1101111) = 104.000 bit
untuk karakter ‘p’ 12.000 x 8bit (1110000) = 96.000 bit
untuk karakter ‘h’ 16.000 x 8bit (1101 000) = 128.000 bit
untuk karakter ‘e’ 9.000 x 8bit (1100101) = 72.000 bit
untuk karakter ‘r’ 5.000 x 8bit (1110010) = 40.000 bit
jumlah = 800.000 bit

3-bit Kode

Karakter	Kode	String Biner
g	0	000
o	1	001
p	2	010
h	3	011
e	4	100
r	5	101

Tabel 7. 3-bit Kode untuk karakter ”g,o,p,h,e,r”

Untuk meng-encoding teks tersebut kita membutuhkan sebanyak

untuk karakter ‘g’ 45.000 x 3 bit (000) = 135.000 bit
untuk karakter ‘o’ 13.000 x 3bit (001) = 39 .000 bit
untuk karakter ‘p’ 12.000 x 3bit (010) = 36 .000 bit
untuk karakter ‘h’ 16.000 x 3bit (011) = 48 .000 bit
untuk karakter ‘e’ 9.000 x 3bit (100) = 27 .000 bit
untuk karakter ‘r’ 5.000 x 3bit (101) = 15.000 bit
jumlah = 300.000 bit

Kode Huffman

Karakter	**Frekuensi**	**Peluang**	**Kode Huffman**
g	45000	3/13	0
o	13000	3/13	101
p	12000	1/13	100
h	16000	1/13	111
e	9000	1/13	1101
r	5000	1/13	1100

Tabel 8. Kode Huffman untuk Karakter “ g,o,p,h,e,r”,

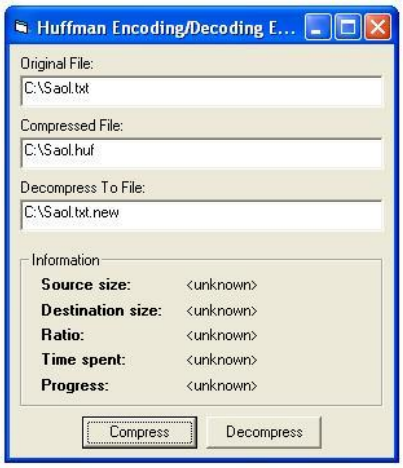
Untuk meng-encoding teks tersebut kita membutuhkan sebanyak

untuk karakter ‘g’ 45.000 x 1 bit (0) = 45 .000 bit
untuk karakter ‘o’ 13.000 x 3bit (101) = 39 .000 bit
untuk karakter ‘p’ 12.000 x 3bit (110) = 36 .000 bit
untuk karakter ‘h’ 16.000 x 3bit (111) = 48.000 bit
untuk karakter ‘e’ 9.000 x 4bit (1101) = 36.000 bit
untuk karakter ‘r’ 5.000 x 4bit (1100) = 20 .000 bit
jumlah = 224.000 bit

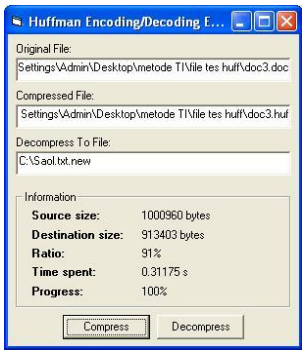
Dari data diatas kita dapat lihat bahwa dengan menggunakan kode ASCII untuk meng-encoding teks tersebut membutuhkan 800.000 bit, sedangkan dengan menggunakan 3-bit kode dibutuhkan 300.000 bit dan dengan menggunakan kode Huffman hanya membutuhkan 224.000. Dengan menggunakan data tersebut maka dapat kita lihat bahwa dengan menggunakan algoritma huffman dapat mengompres teks sebesar 70% dibandingkan kita menggunakan kode ASCII dan sebesar 25,3% dibandingkan kita menggunakan 3-bit kode.

4.1.4 Coding Huffman Compressor

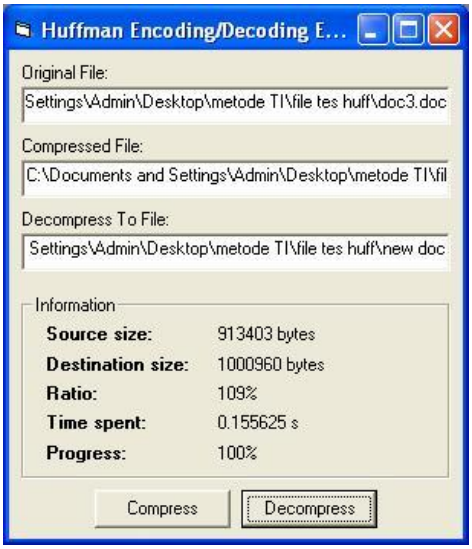
```
'Huffman Encoding/Decoding Class
'-------------------------------
'(c) 2000, Fredrik Qvarfort
'Option Explicit

'Progress Values for the encoding routine
Private Const PROGRESS_CALCFREQUENCY
= 7
Private Const PROGRESS_CALCCRC = 5
Private Const PROGRESS_ENCODING = 88

'Progress Values for the decoding routine
Private Const PROGRESS_DECODING = 89
Private Const PROGRESS_CHECKCRC = 11

'Events
Event Progress(Procent As Integer)

Private Type HUFFMANTREE
  ParentNode As Integer
  RightNode As Integer
  LeftNode As Integer
  Value As Integer
  Weight As Long
End Type

Private Type ByteArray
  Count As Byte
  Data() As Byte
End Type   (*)
```



Gbr 5.1 Huffman



Gbr 5.2 Huffman Compression



Gbr 5.3 Huffman Decompression

4.1.5 Pengujian menggunakan Huffman Compression

File-file yang akan diuji :

- doc (document)
- exe (executable)
- mp3
- txt (teks)

Huffman Compressed

nama file	***source size***	***destination size***	***Ratio***	***time spent***	***progress***
doc1.doc	101376 bytes	81053 bytes	79%	0.046875 s	100%
doc2.doc	516096 bytes	464409 bytes	89%	0.09325 s	100%
doc3.doc	1000960 bytes	913403 bytes	91%	0.24925 s	100%
exe1.exe	1193218 bytes	1188284 bytes	99%	0.328 s	100%
exe2.exe	2309632 bytes	1853533 bytes	80%	0.640125 s	100%
exe3.exe	10296280 bytes	10297059 bytes	100%	2.453125 s	100%
music1.mp3	523476 bytes	510563 bytes	97%	0.390625 s	100%
music2.mp3	1094714 bytes	1085159 bytes	99%	0.49925 s	100%

music3.mp3	5335040 bytes	5320834 bytes	99%	1.21875 s	100%
teks1.txt	109258 bytes	63321 bytes	57%	0.281 s	100%
teks2.txt	546290 bytes	315416 bytes	57%	0.311625 s	100%
teks3.txt	1092580 bytes	630535 bytes	57%	0.40625 s	100%

Huffman Decompressed

nama file	*source size*	*destination size*	*ratio*	*time spent*	*progress*
doc1.doc	81053 bytes	101376 bytes	125%	0.01475 s	100%
doc2.doc	464409 bytes	516096 bytes	111%	0.108875 s	100%
doc3.doc	913403 bytes	1000960 bytes	109%	0.0186875 s	100%
exe1.exe	1188284 bytes	1193218 bytes	100%	0.436875 s	100%
exe2.exe	1853533 bytes	2309632 bytes	124%	0.546125 s	100%
exe3.exe	10297059 bytes	10296280 bytes	124%	0.546125 s	100%
music1.mp3	510563 bytes	523476 bytes	102%	0.187125 s	100%
music2.mp3	1085159 bytes	1094714 bytes	100%	0.202875 s	100%
music3.mp3	5320834 bytes	5335040 bytes	100%	1.093625 s	100%
teks1.txt	63321 bytes	109258 bytes	172%	0.093625 s	100%
teks2.txt	315416 bytes	546290 bytes	173%	0.0625 s	100%
teks3.txt	630535 bytes	1092580 bytes	173%	0.15575 s	100%

Algoritma LZW (Lempel-Ziv-Welch)

4.2.1 Coding LZW Compressor :
Fichier : LZW.PAS

Nombre de lignes : **277 lignes**

```
{$I-,B-,R-,S-}
{Program en Turbo Pascal de
Compression, Algorythme de Lempel-
Ziv-Welch}
const
  CodeClr=256;              {Code de
r,initialisation du Dico}
  CodeEof=257;              {Code de
fin de fichier}
  CodeFirstFree=258;        {Premier
Code Libre}
  MinBits=9;                {Min nb Bits
Code}
  MaxBits=13;               {Max nb
Bits Code}
  MaxCode=1 shl(MaxBits)-1;         {Max
Code}
  NILL=1 shl(MaxBits+1);            {Nil
pointe sur rien}
  TBufFic=1 shl 15;         {Taille
Buffer Fichier}
  TBufPile=1 shl(MaxBits-1);        {Taille
Buffer Pile}
type
  TWord=array[0..MaxCode]of word;
{Tableau Word}
  TByte=array[0..MaxCode]of byte;
{Tableau Byte}
```

TBuffer=array[0..0]of byte; {Buffer}
PBuffer=^TBuffer; {Pointeur Buffer} , ... **

File-file yang akan diuji :

doc (document)
exe (executable)
mp3
txt (teks)

4.2.2 Pengujian menggunakan Huffman Compression

LZW Compressed

nama file	*source size*	*destination size*	*ratio*	*time spent*	*progress*
doc1.doc	**101376 bytes**	**80025 bytes**	**78%**	**1-2 s**	**100%**
doc2.doc	**516096 bytes**	**483713 bytes**	**93%**	**1-2 s**	**100%**
doc3.doc	**1000960 bytes**	**953792 bytes**	**95%**	**1-2 s**	**100%**
exe1.exe	**1193218 bytes**	**1610754 bytes**	**134%**	**1-2 s**	**100%**
exe2.exe	**2309632 bytes**	**1380655 bytes**	**59%**	**1-2 s**	**100%**
exe3.exe	**10296280 bytes**	**14768053 bytes**	**143%**	**24 s**	**100%**
music1.mp3	**523476 bytes**	**684832 bytes**	**130%**	**1-2 s**	**100%**
music2.mp3	**1094714 bytes**	**14778255 bytes**	**135%**	**1-2 s**	**100%**
music3.mp3	**5335040 bytes**	**7429414 bytes**	**139%**	**10 s**	**100%**
teks1.txt	**109258 bytes**	**29592 bytes**	**27%**	**1-2 s**	**100%**
teks2.txt	**546290 bytes**	**148689 bytes**	**27%**	**1-2 s**	**100%**
teks3.txt	**1092580 bytes**	**298092 bytes**	**27%**	**1-2 s**	**100%**

LZW Decompressed

nama file	*source size*	*destination size*	*ratio*	*time spent*	*progress*
doc1.doc	**80025 bytes**	**101376 bytes**	**126%**	**1-2 s**	**100%**
doc2.doc	**483713 bytes**	**516096 bytes**	**106%**	**1-2 s**	**100%**
doc3.doc	**953792 bytes**	**1000960 bytes**	**104%**	**1-2 s**	**100%**
exe1.exe	**1610754 bytes**	**1193218 bytes**	**74%**	**1-2 s**	**100%**
exe2.exe	**1380655 bytes**	**2309632 bytes**	**167%**	**1-2 s**	**100%**
exe3.exe	**14768053 bytes**	**10296280 bytes**	**69%**	**15 s**	**100%**
music1.mp3	**684832 bytes**	**523476 bytes**	**76%**	**1-2 s**	**100%**

music2.mp3	**1478255 bytes**	**1094714 bytes**	**74%**	**1-2 s**	**100%**
music3.mp3	**7429414 bytes**	**5335040 bytes**	**71%**	**10 s**	**100%**
teks1.txt	**29592 bytes**	**109258 bytes**	**369%**	**1-2 s**	**100%**
teks2.txt	**148689 bytes**	**546290 bytes**	**367%**	**1-2 s**	**100%**
teks3.txt	**298092 bytes**	**1092580 bytes**	**366%**	**1-2 s**	**100%**

4.3 Algoritma DMC (Dynamic Markov Compression)

4.3.1 Coding DMC Compression

```
/*   Dynamic Markov Compression (DMC)
Version 0.0.0

#include <stdio.h>

float predict();
int pinit();
int pupdate();

int memsize = 0x1000000;

main(argc,argv)
  int argc;
  char *argv[];
{
  if (argc == 3 && isdigit(*argv[2]))
sscanf(argv[2],"%d",&memsize);
  if (argc >= 2 && *argv[1] == 'c') comp();
  else if (argc >= 2 && *argv[1] == 'e') exp();
  else {
    fprintf(stderr,"usage:  dmc [ce] memsize
<infile >outfile\n");
    exit(1);
  }
  return 0;
}

comp(){
  int max = 0x1000000,
     min = 0,
     mid,
     c,i,
     inbytes = 0,
     outbytes =3,
     pout = 3,
     bit;

(type for spc file)
```

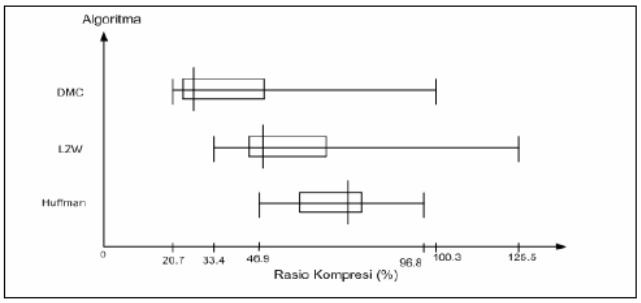
4.3.2 Pengujian menggunakan DMC Compression

nama file	source size	destination size	ratio	time spent
doc1.doc	102400 bytes	4096 bytes	1.333333	5 s
doc2.doc	516096 bytes	4096 bytes	1.333333	5 s
doc3.doc	1003520 bytes	4096 bytes	1.333333	5 s
exe1.exe	1176 bytes	487 bytes	0.413062	1 s
exe2.exe	1179 bytes	487 bytes	0.413062	1 s
exe3.exe	0.413062	487 bytes	0.413062	1 s

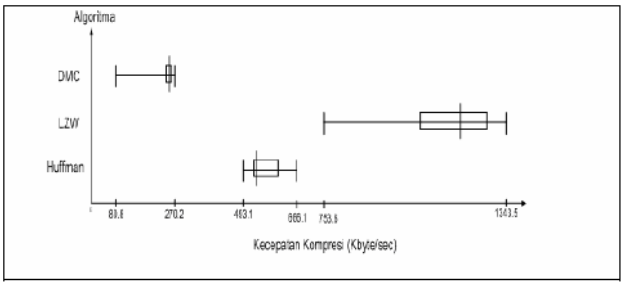
music1.mp3	3100 bytes	523 bytes	0.16871	7 s
music2.mp3	4020 bytes	864 bytes	0.212438	7 s
music3.mp3	8442 bytes	616 bytes	0.072968	7 s
teks1.txt	109258 bytes	12288 bytes	0.622672	5 s
teks2.txt	531440 bytes	330913 bytes	0.622672	5 s
teks3.txt	393216 bytes	219656 bytes	0.558614	5 s

V. KESIMPULAN

Hasil pengukuran statistik terhadap rasio hasil kompresi dan kecepatan kompresi dari ketiga metode di atas pada semua kasus uji dirangkum dalam bentuk *box plot* pada Gambar di bawah ini. Grafik perbandingan rasio kompresi dan kecepatan kompresi dari ketiga metode tersebut dalam masing-masing kasus uji ditunjukkan secara lengkap dalam Gambar 8 dan 9.



Gambar 8. *Box Plot* Rasio Kompresi Ketiga Algoritma



Gambar 9. *Box Plot* Kecepatan Kompresi Ketiga Algoritma

Dari penelitian ini dapat disimpulkan beberapa hal mengenai perbandingan kinerja ketiga metode kompresi yang telah diimplementasikan, yaitu :

1) Secara rata-rata algoritma Huffman menghasilkan rasio file ± 57% - 100%, diikuti algoritma LZW ± 27% - 139% dan terakhir algoritma DMC ± 7% - 133% tetapi untuk DMC hampir semua hasil file tidak dapat di decompressi (file rusak).
2) Untuk kategori file teks dan document, LZW memberikan hasil kompresi yang baik. Sedangkan untuk file multimedia, hasil kompresinya buruk (dapat > 100 %), karena pada umumnya file multimedia merupakan file hasil kompresi juga, dan hasil kompresi LZW terhadap file yang telah terkompresi sebelumnya memang kurang baik.
3) Hasil kompresi Huffman lebih baik dibandingkan LZW hanya pada kasus file biner, file multimedia, dan file executable. Algoritma Huffman memberikan hasil yang relatif hampir sama untuk setiap kasus uji, sedangkan LZW memberikan hasil kompresi yang buruk (dapat > 100%) untuk file multimedia dan file executable.

Daftar Pustaka

http://www.marknelson.us/1989/10/01/lzw-data-compression/
http://www.dspguide.com/ch27/5.htm
http://www.martinreddy.net/gfx/2d/GIF-comp.txt
http://www.notesandsketches.co.uk/Compression_Moulding_DMC.htm
http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=515497
http://comjnl.oxfordjournals.org/cgi/content/abstract/32/1/16

http://www.fel.fujitsu.com/isv-scanner/soft_desc.asp?soft_id=117
http://plg1.cs.uwaterloo.ca/~ftp/dmc/dmc.c
http://www.codeproject.com/cpp/LZWCompression.asp
http://en.wikipedia.org/wiki/LZW
http://www.cs.ucf.edu/courses/cap5015/DMC.pdf
http://www.catenary.com/appnotes/lzwcomp.html
http://libungif.sourceforge.net/doc/lzgif.txt
http://www.leadtools.com/SDK/RASTER/FORMATS/Raster-Format-GIF-TIFLZW.htm
http://cristal.inria.fr/~simonet/teaching/caml-prepa/tp-caml-2001-07.pdf
http://eupat.ffii.org/pikta/xrani/gif-lzw/index.en.html
http://plg1.cs.uwaterloo.ca/~ftp/dmc/
http://plg1.cs.uwaterloo.ca/~ftp/dmc/huffman.txt
http://www.maximumcompression.com/algoritms.php
http://searchsmb.techtarget.com/sDefinition/0,,sid44_gci214337,00.html
http://en.wikipedia.org/wiki/Dynamic_Markov_Compression
http://www.compression-links.info/SourceCode
http://www.wholetomato.com/downloads/downloadTrial.asp
http://www.cbloom.com/src/ppmz.html
http://www.geocities.com/malbrain/vitter_c.html
http://alexvn.freeservers.com/s1/huffman_template_algorithm.html#label_Download
http://en.wikipedia.org/wiki/Data_compression
http://pami.uwaterloo.ca/~mmkd/MMKD07-Cormack.pdf
http://en.wikipedia.org/wiki/LZX_%28algorithm%29
http://www.youtube.com/watch?v=xC5uEe5OzNQ
http://www.google.com/search?q=linux+ubuntu&rls=com.microsoft:en-us:IE-searchBox&ie=UTF-8&oe=UTF-8&sourceid=ie7&rlz=1I7WZPA
http://www.nvidia.com/Download/index.aspx?lang=en-us
http://www.ubuntu.com/getubuntu/downloading?release=desktop-ewest&arch=sparc&mirror=ftp%3A%2F%2Fdl2.foss-d.web.id%2Fiso%2Fubuntu%2Freleases%2F&debug=%5B%27country_US%27%2C+%2country_UK%27%2C+%27continent_NA%27%5D&download-button=%3CIMG+alt%3Ddownload+src%3D%22%2Fthemes%2Fubuntu07%2Fimages%2button-download-new.png%22%3E+Start+Download
http://linux.softpedia.com/get/Multimedia/Graphics/Beryl-19790.shtml
http://linux.softpedia.com/get/System/Operating-Systems/Linux-Distributions/Uberyl-26171.shtml
http://lifehacker.com/software/featured-linux-download/awn-linux-application-dock-64328.php
http://www.linuxnarede.com.br/downloads/
http://www.stttelkom.ac.id/staf/FAY/kuliah/DAA/20052/Tugas1/pdfs/24-DAA%2020052%20113030022%20113030051%20113030301%20Kompresi%20Teks%20dengan%20Menggunakan%20Algoritma%20Huffman.pdf
http://komputasi.inn.bppt.go.id/semiloka05/1805.pdf
http://www.cprogramming.com/tutorial/computersciencetheory/huffman.html
http://en.wikipedia.org/wiki/Huffman_coding
http://www.data-compression.com/lossless.shtml
Algoritma Hufman Menurut :
Irwan Wardoyo, Peri Kusdinar, Irvan Hasbi
Aa greedy Menurut
Linawati dan Henry P. Panggabean lgorit
Algoritma kompresi huffman
Agung W. Setiawan, Andriyan B. Suksmono, dan Tati L.R. Mengko
Algoritma LZW Menurut
Linawati dan Henry P. Panggabean
Algoritma DMC Menurut
Linawati dan Henry P. Panggabean