Konstruksi Kode dengan Redundansi Minimum Menggunakan *Huffman Coding* dan *Range Coding*

Aris Feryanto (NIM: 13507110)
Jurusan Teknik Informatika ITB, Bandung 40132, email: aris_feryanto@yahoo.com

Abstract – *Banyak metode telah ditemukan untuk membuat kode dengan redundansi minimum. Huffman coding dan range coding adalah dua metode dengan pendekatan yang berbeda untuk menghasilkan kode dengan redundansi minimum. Range coding lebih mendekati entropy encoding yang optimal dibandingkan Huffman coding.*

Kata Kunci*: lossless data compresion, Huffman coding, Range coding, entropy encoding.*

1. PENDAHULUAN

Dalam pengiriman dan penyimpanan data digital, ada satu hal yang penting, yaitu mengirimkan atau menyimpan urutan simbol-simbol dari data. Untuk mengirimkan atau menyimpannya, dibutuhkan waktu dan ruang yang sebanding dengan banyaknya simbol yang ada pada data. Karena kebutuhan manusia untuk mengirimkan atau menyimpan data dalam waktu yang cepat dan menggunakan ruang yang sedikit, maka berkembanglah metode-metode yang digunakan untuk meminimalkan redundansi (simbol-simbol yang tidak mengandung informasi) dalam data.

Meminimalkan redundansi dalam data sama saja dengen melakukan kompresi atau pemampatan terhadap data. Pada dasarnya, hal ini dilakukan dengan menggantikan simbol yang probabilitas kemunculannya tinggi dengan simbol yang lebih pendek dan simbol yang probabilitas kemunculannya rendah dengan simbol yang lebih panjang. Dalam makalah ini, akan dibahas dua metode kompresi data, yaitu *Huffmand coding* dan *Range coding*. Keduanya mempunyai pendekatan yang berbeda. *Huffman coding* menggantikan tiap simbol dengan simbol lain yang ditentukan berdasarkan probabilitas kemunculan tiap simbol dari sebuah data. Sedangkan *Range coding*, secara konsep menggantikan keseluruhan simbol dengan sebuah angka.

2. *ENTROPY ENCODING*

Huffman coding dan *Range coding* adalah dua buah algoritma yang populer dalam *entropy encoding*. *Entropy encoding* adalah skema *lossless data compression* yang independen dari karakteristik media penyimpanan. Salah satu jenis dari *entropy coding* adalah dengan membentuk kode prefiks untuk tiap simbol yang muncul. Algoritma tipe ini kemudian menggantikan tiap simbol masukan dengan kode prefiks dengan panjang yang berbeda-beda. Panjang dari kode untuk tiap simbol kurang lebih proporsional dengan negatif dari logaritma probabilitas kemunculan simbol tersebut.

Sesuai dengan teorema pengkodean Shannon (*Shannon's source coding theorem*), *entropy encoding* yang optimal terjadi apabila panjang kode untuk setiap simbol adalah $-\log_2 P$, dimana P adalah probabilitas kemunculan simbol yang bersangkutan.

2.1. *Huffman coding*

Huffman coding dikembangkan oleh David A. Huffman sewaktu dia menjadi mahasiswa Ph.D. di Massachusetts Institute of Technology (MIT), dan dipublikasikan pada tahun 1952 dalam makalah "*A Method for the Construction of Minimum-Redundancy Codes*" [2].

Metode ini menggunakan ketentuan khusus untuk merepresentasikan masing-masing simbol, yaitu kode yang dihasilkan untuk merepresentasikan sebuah simbol tidak membutuhkan indikator untuk menandai dimana bagian awal dan akhir dari masing-masing simbol dalam untaian simbol. Ketentuan ini menghasilkan apa yang disebut dengan "kode Huffman" atau kode prefiks, yaitu kumpulan kode dimana tidak ada kode yang merupakan awalan atau prefiks kode yang lain. Sebagai contoh, kode 00, 01, dan 101 merupakan kode prefiks karena tidak ada sebuah kode yang merupakan awal kode yang lain (misalnya, 01 bukan awal dari kode **10**1), sedangkan kode 00, 01, dan 001 bukan kode prefiks, karena terdapat kode 00 yang merupakan awal dari kode yang lain, yaitu **00**1.

Huffman telah membuktikan bahwa tidak ada pemetaan kode sebuah simbol yang akan menghasilkan rata-rata panjang kode yang lebih pendek daripada kode yang dihasilkan dengan ketentuan di atas.

2.2. Teknik *Huffman coding*

<u>*Encoding*</u>

Pembentukan kode Huffman dapat dilakukan dengan membuat pohon biner. Sebuah simpul (*node*) dalam pohon ini dapat merupakan daun (*leaf*) atau simpul dalam (*internal node*). Pada awalnya, semua simpul merupakan daun yang mengandung simbol yang akan dikodekan dan bobot (probabilitas kemunculan)

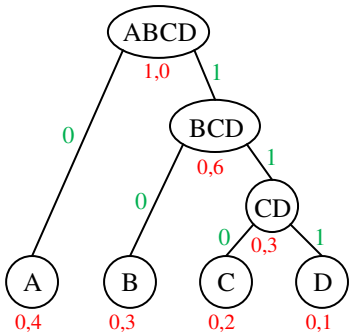
simbol yang bersangkutan. Sebagai konvensi umum, bit '0' merepresentasikan anak kiri dan bit '1' merepresentasikan anak kanan.

Proses dimulai dengan daun-daun yang mengandung probabilitas dari simbol yang direpresentasikan daun tersebut, kemudian simpul baru yang kedua anaknya merupakan simpul dengan probabilitas terkecil dibentuk, dimana probabilitas dari simpul yang baru ini adalah jumlah dari probabilitas kedua anaknya. Dengan menggabungkan kedua simpul menjadi sebuah simpul baru, dan dengan memperhatikan simpul yang baru ini, proses ini diulangi hingga tersisa satu simpul, yaitu pohon Huffman.

Algoritma dari pembentukan pohon Huffman dapat dituliskan sebagai berikut:

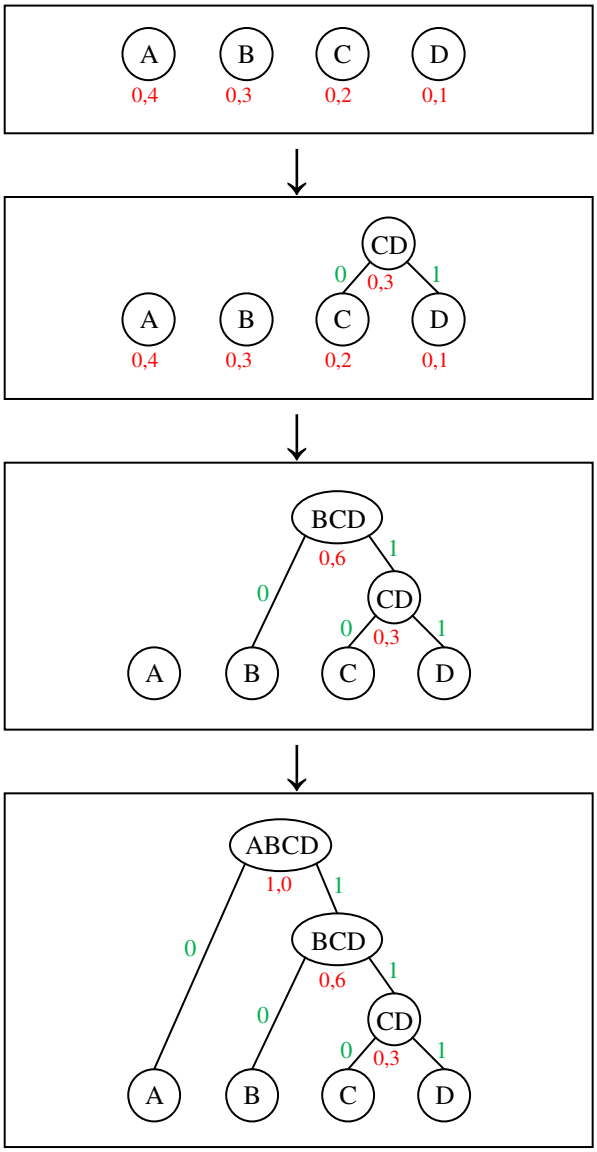
1. Buat daun untuk masing-masing simbol,
2. Urutkan simpul-simpul berdasarkan probabilitas kemunculan,
3. Ambil dua simpul dengan probabilitas terkecil,
4. Buat simpul baru dengan probabilitas yang merupakan jumlah dari probabilitas kedua simpul yang diambil tadi,
5. Dengan memperhatikan simpul yang baru, cek apakah tersisa hanya satu simpul. Jika tidak, maka kembali ke langkah nomor 2, jika ya, maka simpul yang tersisa adalah pohon Huffman dan proses selesai.

Algoritma di atas dapat diimplementasikan dengan mudah menggunakan struktur data *priority queue*. Dengan prioritas berdasarkan pada probabilitas masing-masing simpul. Simpul dengan probabilitas terkecil memiliki prioritas tertinggi.



Gambar 1 : Contoh pohon Huffman untuk 4 simbol berbeda

Contoh:
Misal, kita ingin mengkodekan simbol-simbol "AABDBBAACC". Probabilitas kemunculan masing-masing simbol adalah A:0,4 B:0,3 C:0,2 D:0,1. Untuk menentukan representasi tiap simbol, maka kita harus membangun pohon Huffman dengan cara yang telah disebutkan di atas. Ilustrasi tiap langkahnya adalah sebagai berikut:



Gambar 2 : Langkah pembentukan pohon Huffman

Dengan menggunakan pohon Huffman di atas, kita dapat membuat tabel representasi tiap simbol.

Tabel 1. Representasi bit kode Huffman

Simbol	Representasi Bit
A	0
B	10
C	110
D	111

Dengan demikian, representasi "AABDBBAACC" adalah 0010111101000110110 yang panjangnya 19 bit. Ini jauh lebih pendek dibandingkan panjang bit yang dibutuhkan untuk menyimpan simbol-simbol ini sebagai karakter pada file teks di komputer saat ini. Sebagai perbandingan, pada komputer saat ini, untuk menyimpan sebuah karakter dibutuhkan 8 bit, sehingga untuk menyimpan seluruh simbol "AABDBBAACC" dibutuhkan 10*8 = 80 bit, atau dengan kata lain rasio kompresi kode Huffman untuk simbol-simbol ini adalah 19/80 * 100% = 23,75%.

Namun, dalam implementasinya, representasi tiap simbol yang telah dikodekan (atau pohon Huffman) perlu ditransmisikan atau disimpan agar bisa diubah kembali menjadi simbol-simbol awal, sehingga rasio kompresi yang dicapai berubah sedikit dari 23,75%.

Decoding

Proses *decoding* pada *Huffman coding* adalah kebalikan dari proses *encoding*-nya. Dengan teknik tertentu, representasi bit untuk tiap simbol telah ditransmisikan atau disimpan. Dengan membaca untaian bit masukan satu per satu dan mencocokkannya dengan representasi bit yang sesuai, kita dapat mengembalikan simbol awalnya.

Untuk pohon Huffman yang diberikan, algoritma *decoding* dengan *traversal* yang dimulai dari akar pohon Huffman adalah:

1. Ambil satu bit dari masukan,
2. Berjalan ke anak kiri atau kanan sesuai bit yang didapat dari langkah 1 (sesuai konvensi umum, 0 berarti ke anak kiri, 1 berarti ke anak kanan),
3. Apabila simpul sekarang adalah daun, keluarkan simbol yang direpresentasikan oleh daun tersebut dan *traversal* kembali ke akar,
4. Apabila masukan belum habis, kembali ke langkah 1.

Contoh:

Untuk mengembalikan simbol awal dari untaian bit pada contoh sebelumnya "0010111101000110110", dan dengan pohon Huffman seperti pada gambar 1, maka langkah pertama yang dilakukan adalah mengambil bit paling pertama, yaitu 0. Kemudian berjalan dari akar ke anak kiri (karena bit 0) dan karena simpul ini adalah daun, maka didapatkan simbol pertama adalah A dan *traversal* kembali dimulai dari akar. Hal yang sama dilakukan untuk bit ke-dua. Untuk bit ke-tiga, setelah dibaca dari masukan, berjalan dari akar ke anak kanan (karena bit 1). Karena simpul sekarang bukan daun, baca bit berikutnya, yaitu 0. Dari simpul sekarang, berjalan ke anak kiri. Karena simpul sekarang adalah daun, maka didapatkan simbol yang ke-tiga adalah simbol yang direpresentasikan oleh daun ini, yaitu B dan *traversal* kembali dimulai dari akar. Demikian seterusnya hingga masukan habis.

2.3. *Range coding*

Range coding adalah metode kompresi data yang dirumuskan oleh G N N Martin pada tahun 1979 dalam makalah "*Range encoding: an algorithm for removing redundancy from a digitized message*" [2]. Pada dasarnya, *range coding* sama dengan *arithmetic coding* yang dikembangkan oleh J. J. Rissanen. Perbedaan antara keduanya, yaitu pada rentang angka yang digunakan, akan diperlihatkan pada bagian akhir dari upabab ini.

Tidak seperti *Huffman coding* yang memetakan tiap simbol ke kode tertentu, *range coding* secara konsep mengkodekan seluruh simbol dalam data menjadi sebuah bilangan. Oleh karena itu, *range coding* memberikan rasio kompresi yang lebih besar dibandingkan *Huffman coding*.

Konsep dasar dari *range* coding adalah sebagai berikut. Untuk simbol-simbol dan probabilitas untuk simbol yang bersangkutan yang diberikan, ditentukan sebuah rentang (*range*, oleh sebab itu dinamakan *range coding*) bilangan bulat yang cukup besar. Kemudian rentang tersebut dibagi menjadi N uparentang (N adalah banyaknya simbol) yang proporsional dengan probabilitas kemunculan tiap simbol. Angka-angka di uparentang ini diambil untuk merepresentasikan simbol pertama. Untuk simbol berikutnya, uparentang ini kemudian dibagi lagi menjadi uparentang yang lebih kecil dan proporsional dengan probabilitas kemunculan simbol. Angka-angka pada uparentang yang lebih kecil ini diambil untuk merepresentasikan gabungan simbol pertama dan kedua. Begitu seterusnya untuk simbol-simbol berikutnya.

Dalam *range coding*, rentang yang dipilih umumnya adalah bilangan bulat dari nol hingga suatu bilangan positif. Hal ini yang membedakan *range coding* dengan *arithmetic coding*, karena pada *arithmetic coding*, rentang yang digunakan adalah bilangan pecahan antara 0,0 hingga 1,0.

2.4. Teknik *range coding*

Encoding

Misal a_i adalah simbol ke-i yang akan dikodekan. Kita harus memilih rentang yang cukup besar, misal r, untuk mengkodekan a_1, kemudian sisa rentangnya untuk mengkodekan a_2, kemudian sisa rentang dari a_2 digunakan untuk mengkodekan a_3, demikian seterusnya.

Contoh:

Misalkan kita ingin mengkodekan simbol-simbol "AEAAF". Untuk memperjelas contoh, maka rentang bilangan yang digunakan di sini adalah bilangan basis-10 (untuk komputer saat ini, kita harus mengkodekannya dalam biner), dengan rentang awal adalah [0, 100000), dan tabel frekuensi untuk masing-masing simbol:

Tabel 2. Frekuensi simbol contoh *range coding*

Simbol	Frekuensi
A	0,6
E	0,2
F	0,2

Untuk mengkodekan simbol pertama, maka rentang awal akan dibagi menjadi:

Tabel 3. Rentang simbol pertama contoh *range coding*

Simbol	Rentang
A	[0, 60000)
E	[60000, 80000)
F	[80000, 100000)

Karena simbol pertama adalah A, maka rentang awal dipersempit menjadi rentang dari simbol A, yaitu [0, 60000). Rentang ini yang akan selanjutnya digunakan untuk mengkodekan simbol ke-dua. Rentang ini dibagi menjadi uparentang sebagai berikut:

Tabel 4. Rentang simbol ke-dua contoh *range coding*

Simbol	Rentang
AA	[0, 36000)
AE	[36000, 48000)
AF	[48000, 60000)

Dengan mencocokkan simbol yang ke-dua dengan simbol yang ada pada tabel di atas, maka rentang selanjutnya menjadi [36000, 48000).

Tabel 5. Rentang simbol ke-tiga contoh *range coding*

Simbol	Rentang
AEA	[36000, 43200)
AEE	[43200, 45600)
AEF	[45600, 48000)

Untuk simbol selanjutnya, digunakan rentang yang mewakili simbol-simbol "AEA".

Tabel 6. Rentang simbol ke-empat contoh *range coding*

Simbol	Rentang
AEAA	[36000, 40320)
AEAE	[40320, 41760)
AEAF	[41760, 43200)

Pilihan rentang kita adalah yang pertama dari tiga pilihan di atas. Mungkin sampai pada tahap ini, kita merasa sulit untuk membagi rentang berikutnya. Tapi sebenarnya tidak. Kita bisa mengurangi batas atas rentang dengan batas bawahnya dan mengetahui bahwa ada 4320 bilangan dalam rentang, 2592 mewakili 0.6 bagian awal, 864 mewakili 0.2 bagian selanjutnya, dan 864 mewakili 0.2 bagian terakhir. Dengan menambahkan bilangan-bilangan ini dengan batas bawah rentang, maka di dapatkan:

Tabel 7. Rentang simbol ke-lima contoh *range coding*

Simbol	Rentang
AEAAA	[36000, 38592)
AEAAE	[38592, 39456)
AEAAF	[39456, 40320)

Akhirnya, rentang yang kita dapatkan untuk merepresentasikan simbol-simbol "AEAAF" adalah [39456, 40320). Namun, bila kita lihat, angka dengan panjang 5 digit yang diawali "396" berada pada rentang terakhir kita. Oleh karena itu, cukup dengan menggunakan tiga digit awal tersebut, kita telah dapat merepresentasikan semua simbol "AEAAF".

Decoding

Untuk mengembalikan simbol-simbol yang telah dikodekan, dibutuhkan tabel frekuensi dan rentang awal s yang sama seperti yang digunakan pada saat mengkodekan. Proses yang dilakukan hampir sama dengan saat mengkodekan, hanya saja bukan menentukan pilihan rentang mana yang diambil, namun mencocokkan di rentang mana bilangan yang kita ketahui berada. Untuk rentang awal s, kita mendapatkan simbol pertama a_1, rentang sisanya digunakan untuk mendapatkan simbol ke-dua a_2, sisa dari rentang untuk a_2 digunakan untuk mendapatkan a_3, dan seterusnya.

Contoh:

Dari contoh sebelumnya, kita menggunakan rentang awal [0, 100000), frekuensi seperti pada tabel 2, panjang simbol yang dikodekan, misal L, 5 simbol, dan bilangan yang diketahui adalah bilangan 5-digit yang diawali "396". Misal, kita pilih bilangan X = 39600, untuk menentukan simbol ke-i, a_i ($1 \leq i \leq L$), rentang awal dibagi menjadi:

Tabel 8. Rentang simbol pertama *decoding* contoh *range coding*

Rentang	Simbol
[0, 60000)	A
[60000, 80000)	E
[80000, 100000)	F

Karena X jatuh pada rentang pertama, maka didapatkan a_1 = A. Selanjutnya, rentang [0, 60000) akan digunakan untuk menentukan simbol berikutnya.

Tabel 9. Rentang simbol ke-dua *decoding* contoh *range coding*

Rentang	Simbol
[0, 36000)	AA
[36000, 48000)	AE
[48000, 60000)	AF

Dengan mencocokkan X dengan rentang ke-dua, kita mendapatkan a_2 adalah E.

Tabel 10. Rentang simbol ke-tiga *decoding* contoh *range coding*

Rentang	Simbol
[36000, 43200)	AEA
[43200, 45600)	AEE
[45600, 48000)	AEF

Dari tiga rentang di atas, bilangan yang diketahui berada pada rentang yang pertama, oleh karena itu, didapatkan a_3 adalah A. Untuk rentang selanjutnya [36000, 43200), dibagi menjadi uparentang:

Tabel 11. Rentang simbol ke-empat *decoding* contoh *range coding*

Rentang	Simbol
[36000, 40320)	AEAA
[40320, 41760)	AEAE
[41760, 43200)	AEAF

Karena $36000 \leq X \leq 40320$, maka simbol berikutnya, a_4, adalah A.

Tabel 12. Rentang simbol ke-lima *decoding* contoh *range coding*

Rentang	Simbol
[36000, 38592)	AEAAA
[38592, 39456)	AEAAE
[39456, 40320)	AEAAF

Untuk simbol yang terakhir (kita tahu terakhir karena i sudah sama dengan L), didapatkan a_5 adalah F. Dengan menyatukan semua simbol-simbol, maka didapatkan keseluruhannya adalah "AEAAF". Hasil ini cocok dengan simbol awal pada contoh sebelumnya.

Implementasi

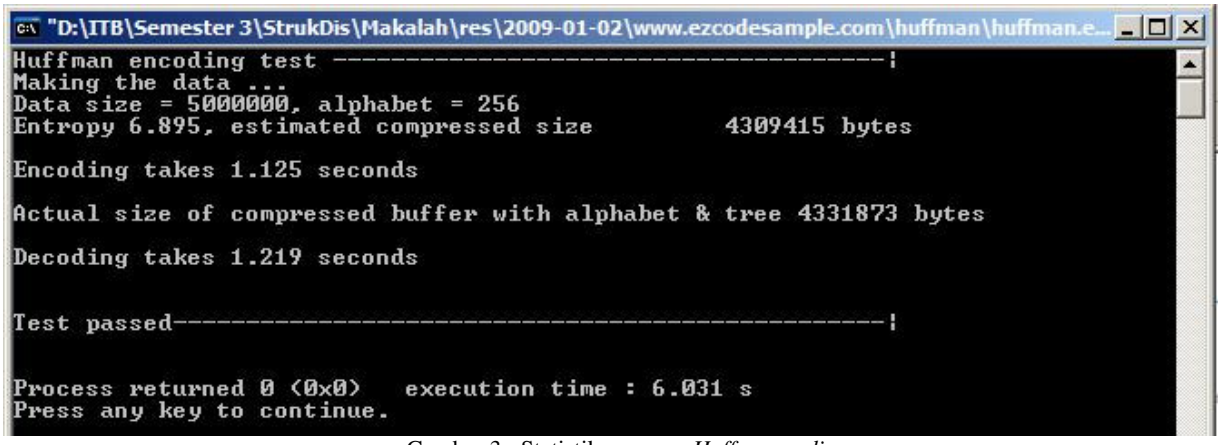
Kendala utama yang dihadapi dalam implementasi algoritma *range encoding* adalah batas nilai bilangan yang dapat direpresentasikan oleh komputer saat ini. Untuk mengatasi terjadinya *overflow*, kita dapat melihat jika awalan batas bawah dan batas atas rentang sudah sama, tidak mungkin awalan ini akan berubah lagi. Sebagai contoh, jika rentang yang kita miliki adalah [25678, 25998), maka untuk rentang berikutnya, pasti memiliki awalan "25". Kita dapat langsung mengirimkan atau menyimpan awalan ini, dan kemudian menggeser (*shift*) bilangan di belakangnya, sehingga rentang menjadi [67800, 99800). Dengan cara ini, *overflow* dapat diatasi.

Ada masalah lain lagi yang timbul, yaitu *underflow*, yang terjadi bila batas atas dan bawah sangat dekat, tapi tidak mempunyai awalan yang sama, misalnya [29951, 30008). Untuk mengatasinya, kita menyimpan awalan di tempat sementara, kemudian menggeser bilangan di belakangnya.

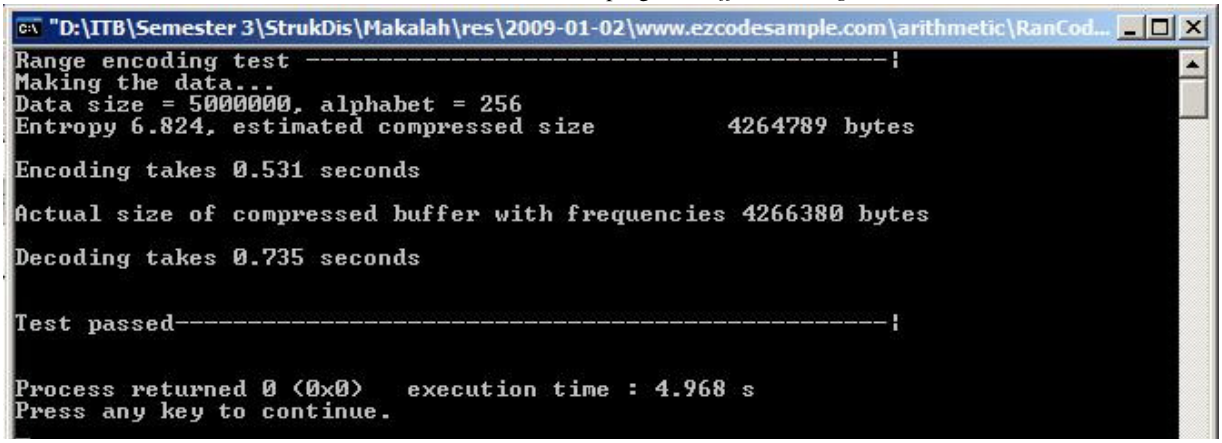
3. PERBANDINGAN *HUFFMAN CODING* DAN *RANGE CODING*

Dengan menggunakan kode sumber *Huffman coding* dan *range coding* dari situs www.ezcodesample.com [8] yang sedikit dimodifikasi, didapatkan hasil berupa waktu yang dibutuhkan untuk melakukan pengkodean dan ukuran kode hasilnya seperti pada gambar 3 dan gambar 4.

Data acak yang digunakan dalam pengkodean diatur sedemikian sehingga berukuran 5000000 bytes. Untuk *Huffman coding*, didapatkan hasil kompresi untuk suatu data acak tersebut adalah sebesar 4331873 bytes



Gambar 3 : Statistik program *Huffman coding*



Gambar 4 : Statistik program *range coding*

(sudah termasuk simbol yang digunakan dan pohon Huffman) dari estimasi *entropy encoding* yang optimal sebesar 4309415 bytes. Atau dengan kata lain, inefisiensi sebesar 4331873 – 4309415 = 22458 bytes.

Sedangkan untuk *range coding*, didapatkan hasil kompresi untuk data masukan yang berbeda dengan ukuran yang sama adalah sebesar 4266380 bytes (sudah termasuk tabel frekuensi yang digunakan) dari estimasi *entropy encoding* yang optimal sebesar 4264789 bytes. Atau dengan kata lain, inefisiensi sebesar 4266380 – 4264789 = 1591 bytes.

Dengan demikian, terbukti bahwa *range coding* lebih mendekati *entropy encoding* yang optimal daripada *Huffman coding*.

Untuk kecepatan pengkodean (*encoding*) dan *decoding*, pada umumnya, *Huffman coding* lebih cepat daripada *range coding*. Namun untuk program pada kode sumber ini, *Huffman coding* membutuhkan waktu yang lebih lama daripada *range coding* karena implementasi struktur dan algoritma pembentukan pohon Huffman yang masih sangat sederhana sehingga waktu operasi menjadi lebih lama.

4. KESIMPULAN

Huffman coding dan *range coding* adalah dua algoritma kompresi yang berbeda dan memiliki kelebihan dan kekurangan masing-masing. Di sisi rasio kompresi, *range coding* lebih mendekati *entropy encoding* yang optimal daripada *Huffman coding*, karena tidak memetakan tiap simbol dengan representasi bit tertentu, melainkan mengubah untaian simbol-simbol menjadi satu bilangan tertentu. Namun, di sisi kecepatan proses, pada umumnya, *Huffman coding* lebih cepat daripada *range coding*.

DAFTAR REFERENSI

[1] Huffman coding – Wikipedia, http://en.wikipedia.org/wiki/Huffman_coding, Tanggal akses: 21 Desember 2008

[2] A Method for the Construction of Minimum Redundancy Codes, D. A. Huffman, Proc. I.R.E., vol 4D, no. 9, 1952, pp 1098-1101

[3] Advanced tricks about Huffman, http://www.compressconsult.com/huffman, Tanggal akses: 29 Desember 2008

[4] Range encoding – Wikipedia, http://en.wikipedia.org/wiki/Range_encoding, Tanggal akses: 27 Desember 2008

[5] Range Encoding: an Algorithm for Removing Redundancy from a Digitised Message, G. N. N. Martin, IBM UK Scientific Center

[6] Range coder by Arturo Campos, http://www.arturocampos.com/ac_range.html, Tanggal akses: 28 Desember 2008

[7] A version of arithmetic coding, http://www.compressconsult.com/rangecoder, Tanggal akses: 29 Desember 2008

[8] Kode sumber Huffman coding dan range coding, http://www.ezcodesample.com/, Tanggal akses: 2 Januari 2009

[9] Arithmetic coding – Wikipedia, http://en.wikipedia.org/wiki/Arithmetic_coding, Tanggal akses: 22 Desember 2008

[10] Entropy encoding – Wikipedia, http://en.wikipedia.org/wiki/Entropy_encoding, Tanggal akses: 21 Desember 2008