PENINGKATAN EFISIENSI KODE HUFFMAN *(HUFFMAN CODE)* DENGAN MENGGUNAKAN KODE HUFFMAN KANONIK *(CANONICAL HUFFMAN CODE)*

Rd. Aditya Satrya Wibawa – NIM : 13505064

Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : if15064@students.if.itb.ac.id

Abstrak

Teknik pemampatan data dengan Kode Huffman menghasilkan penghematan hingga tiga puluh persen. Namun, masih terdapat kelemahan-kelemahan dari kode Huffman sehingga terdapat inefisiensi pada beberapa hal. Kelemahan-kelemahan dalam hal efisiensi yang terdapat pada kode Huffman adalah seperti ruang penyimpanan *codebook* yang besar, lambatnya proses *decoding*, serta waktu yang dibutuhkan untuk melakukan sinkronisasi terhadap kesalahan (*error*) relatif panjang.
Makalah ini membahas tentang peningkatan efisiensi proses *decoding* kode Huffman *(Huffman Code)* dengan menggunakan kode Huffman kanonik *(Canonical Huffman Code)* Kode Huffman kanonik merupakan salah satu bentuk variasi dari kode Huffman yang lebih efisien. Beberapa keuntungannya adalah ruang penyimpanan *codebook* yang relatif kecil, sinkronisasi terhadap kesalahan (*error*) relatif cepat, serta efisiennya proses *decoding*, dan ini merupakan keuntungan yang paling signifikan dari kode Huffman kanonik.

Kata kunci:. *kode Huffman, kode Huffman kanonik, efisiensi, encoding, decoding, pohon Huffman, pohon kanonik, tabel kanonik*

1. Pendahuluan

Data yang berukuran besar seringkali sulit untuk disimpan dan. 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 mengompresi teks.agar ukurannya menjadi lebih kecil dari semula
Kompresi adalah proses pengubahan sekumpulan data menjadi bentuk kode dengan tujuan untuk menghemat kebutuhan tempat penyimpanan dan waktu untuk transmisi data [2]. Dengan menggunakan algoritma Huffman, proses pengompresan teks dilakukan dengan menggunakan prinsip pengkodean, yaitu tiap karakter dikodekan menghasilkan hasil yang lebih optimal. Teknik ini mampu memampatkan sampai dengan tiga puluh persen dari ukuran semula. Tetapi proses *decoding string* biner menjadi data kembali masih kurang efisien. Karena itu, kami mencoba membahas salah satu algoritma yang mirip dengan algoritma Huffman tetapi lebih efisien yaitu algoritma Huffman kanonik.

2. Kode Huffman

2.1 Deskripsi Umum

Dalam ilmu komputer dan teori informasi, kode Huffman merupakan algoritma pengkodean yang optimal yang digunakan dalam pemampatan data. Kode Huffman menggunakan sebuah tabel variasi panjang kode untuk melakukan *encoding* sebuah simbol (misalnya karakter dalam file) dimana tabel panjang kode sudah dibangun sedemikian berdasarkan peluang kemunculan tiap simbol dalam suatu file. Kode ini dikembangkan oleh David A. Huffman saat ia sedang menjalani pendidikan doktoralnya di MIT, dan dipublikasikan dalam papernya “A Method for the Construction of Minimum-Redundancy Codes” pada tahun 1952. Huffman menjadi anggota MIT faculty begitu lulus. dan di kemudian hari menjadi salah satu pendiri Departemen Ilmu Komputer Universitas Calofornia, Santa Cruz. yang sekarang menjadi bagian dari Baskin School of Engineering.
Kode Huffman menggunakan metode yang spesifik untuk merepresentasikan setiap simbol, yang dihasilkan dalam sebuah prefix-free code yang merepresentasikan simbol yang paling

lebih sering muncul dengan dengan string dengan jumlah bit lebih pendek daripada simbol yang kemunculannya lebih jarang [8].

2.2. Sejarah

Pada tahun 1951, David Huffman dan teman sekelasnya di jurusan Teori Infoemasi MIT diberikan pilihan untuk membuat paper ketimbang ujian akhir. Professornya, Robert M. Fano, menugaskan sebuah tema paper dalam permasalahan mencari kode biner yang paling efisien. Huffman tidak dapat membuktikan kode manapun sebagai kode yang paling efisien. Dia hampir saja menyerah dan akan mulai belajar untuk ujuan akhirnya. Namun, ternyata dia mendapatkan ide untuk menggunakan sebuah pohon biner yang dibangun berdasarkan pengurutan frekuensi dan dengan cepan dibuktikannya bahwa metode ini adalah yang paling efisien [8].

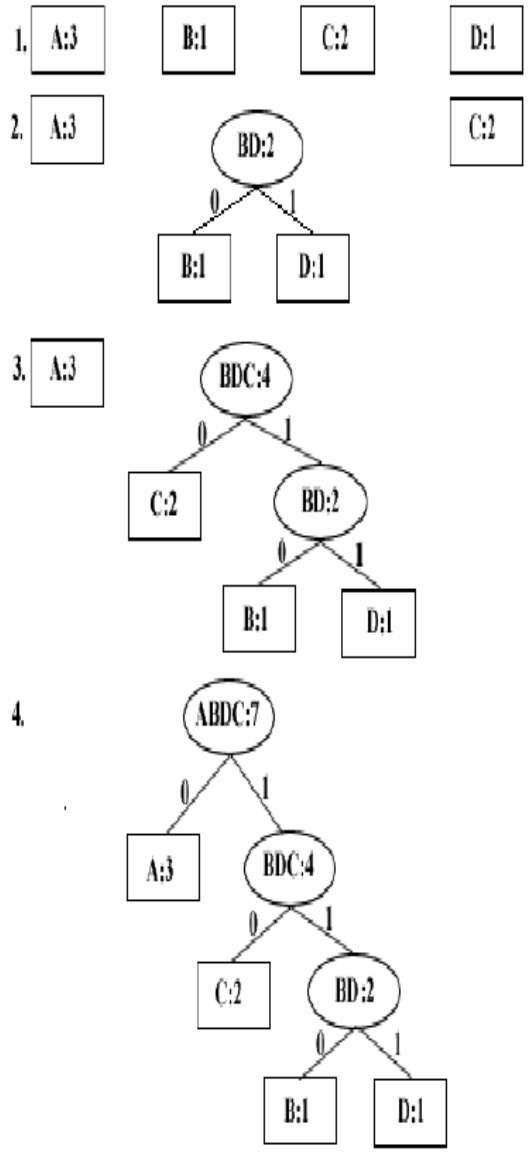
2.3. Pembentukan Pohon Huffman

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.
Langkah-langkah pembentukan pohon Huffman adalah sebagai berikut [2] :

1. Baca semua karakter di dalam teks untuk menghitung frekuensi kemunculan setiap karakter. Setiap karakter penyusun teks dinyatakan sebagai pohon bersimpul tunggal. Setiap simpul di-*assign* dengan frekuensi kemunculan karakter tersebut.
2. Terapkan strategi algoritma *greedy* sebagai berikut : gabungkan dua buah pohon yang mempunyai frekuensi terkecil pada sebuah akar. Setelah digabungkan akar tersebut akan mempunyai frekuensi yang merupakan jumlah dari frekuensi dua buah pohon-pohon penyusunnya.
3. Ulangi langkah 2 sampai hanya tersisa satu buah pohon Huffman. Agar pemilihan dua pohon yang akan digabungkan berlangsung cepat, maka semua yang ada selalu terurut menaik berdasarkan frekuensi.

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



Gambar 1. Pohon Huffman untuk Karakter "ABACCDA" [2]

2.4. 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 [4]

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*

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

Simbol	**Kode Huffman**
A	0
B	110
C	10
D	111

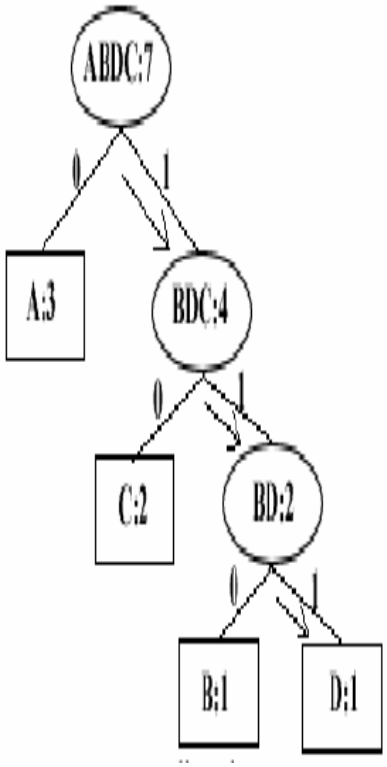
Tabel 1. Kode Huffman untuk karakter "ABCD"

2.5. 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 : [4]

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"



Gambar 2. Proses Decoding dengan menggunakan pohon Huffman [4]

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 1 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 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 lalu baca lagi kode bit berikutnya, sehingga menjadi "110". Rangkaian kode bit "110" adalah pemetaan dari simbol "B".

2.6. Kelemahan Kode Huffman dan Penggunaan Kode Huffman Kanonik untuk Mengatasinya

Kode Huffman memiliki beberapa kelemahan utama. Terutama jika ukuran alfabet besar. Kelemahan tersebut adalah: [6]

1. Space: jika terdapat n simbol berbeda, maka terdapat n simpul daun dan n-1 simpul dalam. Setiap simpul dalam memiliki dua pointer, yang menunjuk ke anak kanan da anak kirinya. Setiap simpul daun memiliki pointer yang menunjuk ke nilai simbol yang dimaksud dan sebuah flag yang menandakan bahwa dia adalah simpul daun. Jadi, sebuah pohon Huffman membutuhkan space memori sekitar 4n *words*.
2. Proses *decoding* yang lambat, membutuhkan proses traversal pada keseluruhan pohon dengan banyak pointer dan tanpa adanya akses penyimpanan yang terlokalisasi. Setiap bit membutuhkan akses memori selama proses *decoding*.

Untuk mengatasi kelemahan-kelemahan dari kode Huffman, dapat digunakan sebuah variasi dari kode Huffman, yaitu kode Huffman Kanonik (*Canonical Huffman Code*) yang tidak membutuhkan proses traversal yang panjang, hanya perlu sedikit akses terhadap memori, dan juga hanya memerlukan ruang penyimpanan (*storage*) yang lebih sedikit dibandingkan kode Huffman [6]. Kode Huffman Kanonik sangat bermanfaat saat ukuran alfabet besar tetapi kita membutuhkan proses *decoding* yang cepat.
Kode disimpan dalam alamat memori yang kontigu bersama dengan simbol. Proses encoding dan, terutama, *decoding*-nya sangat cepat. Kode ini didesain sedemikian sehingga kita hanya memerlukan panjang kode (*codelength*) sebagai masukan bagi *encoder*. Langkah-langkah pengkodean Huffman kanonik membutuhkan proses komputasi yang lebih banyak dibandingkan kode Huffman, namun tetap lebih efisien karena jauh lebih sedikit membutuhkan akses terhadap memori.
Pada bagian selenjutnya dari makalah ini, akan kami jelaskan lebih detail menegenai kode Huffman kanonik.

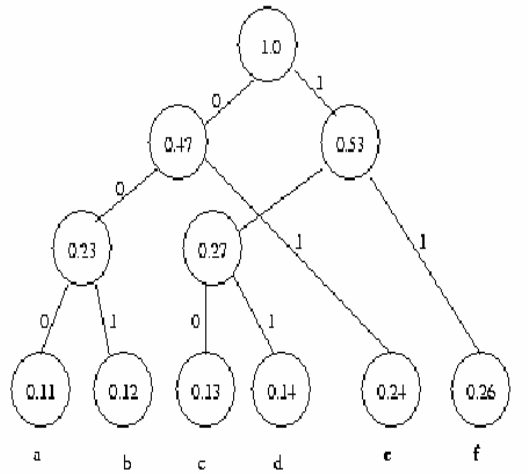
3. Kode Huffman Kanonik

2.1. Deskripsi Umum

Kode Huffman kanonik merupakan suatu bentuk variasi dari kode Huffman yang memiliki sifat dapat dideskripsikan dengan sangat mampat (*compactly described*) [9]. Hal ini disebabkan kode Huffman kanonik memiliki aturan-aturan penulisan yang baku sehingga beberapa hal dapat disepakati dan tidak perlu dituliskan pada deskripsi kode. Contohnya adalah kode harus terurut berdasarkan urutan simbol, panjang kode suatu simbol harus sesuai dengan aras simpul simbol tersebut pada pohon Huffman, dll. Deskripsi kode yang *compact* membuat kode ini lebih efisien dalam bberapa hal dibandingkan dengan kode Huffman.

2.2. Keunikan Kode Huffman Kanonik [6]

Perhatikan contoh dengan suatu distribusi peluang kemunculan karakter seperti di bawah ini :



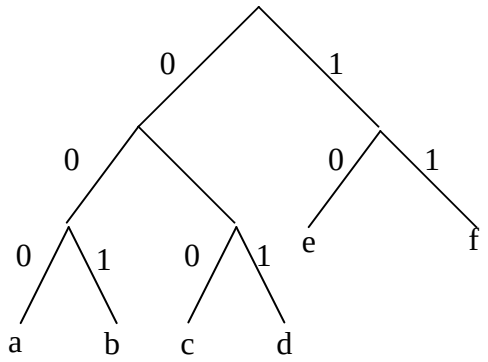
Gambar 3. Pohon Huffman

Simbol	Kode
a	000
b	001
c	110
d	111
e	01
f	11

Tabel 2. Kode Huffman dari Pohon Huffman pada gambar 3

Seperti yang kita ketahui, jika terdapat n-1 simpul dalam, kita dapat membuat 2^{n-1} kode Huffman baru dengan cara melabeli ulang

(*relabelling*) setiap sisi dari simpul dalam (karena setiap sisi simpul dalam memiliki dua pilihan kemungkinan pelabelan, yaitu 0 dan 1). Pada contoh di atas kita akan memiliki $2^5 = 32$ kode Huffman yang berbeda. Tetapi, mari kita nyatakan kode tersebut dalam sebagai 00x, 10y, C, D. Semua permutasi yang mungkin terhadap A, B, C, dan D akan menghasilkan kode Huffman yang valid dalam arti panjang kodenya (*codelength*) akan sama dan semua kodenya akan berupa kode awalan (*prefix code*). Terdapat 4! permutasi yang mungkin dan (x,y) memiliki empat kemungkinan nilai. Maka, total kode Huffman yang mungkin dihasilkan sebenarnya berjumlah 96 buah kode. Ini berarti terdapat kode-kode awalan lain yang tidak bisa dihasilkan oleh pohon Huffman (*Huffman tree*), tetapi memiliki panjang kode rata-rata (*average codelength*) yang sama dengan yang dihasilkan oleh pengkodean Huffman. Kode Huffman kanonik (*Canonical Huffman Code*) merupakan kode yang ”semacam” itu. Sebuah contoh diberikan berikut ini



Gambar 4. Pohon Huffman knonik

Simbol	Kode
a	000
b	001
c	010
d	011
e	10
f	11

Tabel 3. Kode Huffman dari Pohon Huffman pada gambar 4

Pohon biner yng ditunjukkan di atas tidak dapat diturunkan melalui algoritma Huffman. Tetapi, pohon diatas merupakan pohon kode prefiks, memiliki redundansi minimal, dan memiliki panjang kode rata-rata yang sama dengan yang dihasilkan algoritma Huffman

2.3. Pembentukan Pohon Huffman

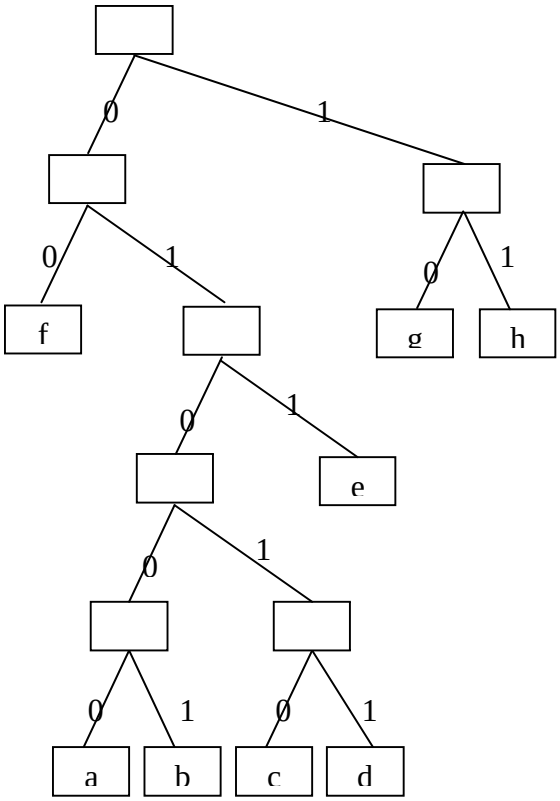
Pembentukan pohon Huffman pada algoritma Huffman Kanonik hampir sama dengan pembentukan pohon Huffman pada algoritma Huffman biasa. Bedanya adalah pada algoritma Huffman Kanonik, pohon Huffman-nya tidak diberi label 0 atau 1 pada sisi-sisinya.

2.4. Proses *Encoding*

Aturan-aturan proses *encoding* pada algoritma Huffman Kanonik adalah sebagai berikut:

1. Panjang kode untuk suatu simpul adalah sebesar aras simpul tersebut.
2. *String* biner simpul paling dalam yang terletak paling kiri diberi nilai 0 semuanya. Untuk simpul berikutnya(bergeser dari kiri ke kanan) *string* binernya naik satu nilai dari simpul sebelumnya.
3. Apabila semua simpul pada kedalaman yang sama telah di-*encode*, maka proses *encoding* dilanjutkan ke aras yang lebih rendah. Hanya saja *string* binernya tidak dimulai dengan semuanya 0. String biner simpul paling kiri dimulai dengan kode baru. Kode baru itu merupakan kenaikan 1 nilai dari *string* biner simpul yang terakhir di-*encode* namun biner paling belakangnya dihilangkan sehingga panjang kodenya berkurang satu. Simpul berikutnya di*encode* dengan *string* binernya naik satu nilai(bergeser dari kiri ke kanan).
4. Proses berhenti bila telah mencapai akar.

Untuk lebih jelasnya, mari kita perhatikan contoh berikut ini. Misalkan pada contoh ini, terdapat suatu distribusi peluan kemunculan karakter sedemikian sehingga menghasilkan pohon Huffman seperti di bawah ini.



Gambar 5. Pohon Huffman kanonik

Pada pohon Huffman tersebut, karakter ‘a’ terletak pada simpul paling dalam dan paling kiri dan diberi kode 00000. Karakter ‘b’ terletak di sebelah kanan ‘B’ pada aras yang sama sehingga diberi kode 00001 yang lebih satu nilai dari 00000. Lalu karakter ‘c’ dan ‘d’ diberi kode 00010 dan 00011 dengan cara yang sama. Untuk karakter ‘e’ yang berada pada dua aras yang lebih rendah diberi kode yang lebih satu nilai dari 00011 yaitu 00100 namun dua biner di belakangnya dihilangkan menjadi 001. Untuk karakter ‘f’ yang berada pada aras yang lebih rendah diberi kode yang lebih satu nilai dari 001 yaitu 010 namun satu biner di belakangnya dihilangkan menjadi 01. Karakter ‘g’ yang berada pada aras yang sama dengan ‘f’ diberi kode 10 yang lebih satu nilai dari 01. Lalu karakter ‘h’ diberi kode 11 dengan cara yang sama. Demikian seterusnya hingga semua karakter telah di-*encode*.

Dari langkah-langkah di atas, kita dapat merumuskannya dalam tabel berikut ini.

Simbol	**Kode Huffman Kanonik**	**Kode Huffman ”biasa”**	**Aras**
a	00000	01000	5
b	00001	01001	5
c	00010	01010	5
d	00011	01011	5
e	001	011	3
f	01	00	2
g	10	10	2
h	11	11	2

Tabel 4. Perbandingan kode yang dihasilkan antara kode Huffman dengan kode huffman kanonik

Langkah-langkah di atas mengatarkan kita pada sebuah kode Huffman Kanonik yang dapat dilihat pada tabel di atas. Namun, langkah-langkah trsebut belumlah cukup untuk membuat kode Huffman yang efisien pada saat akan di-*decode*. Suatu tabel kanonik harus dibuat terlebih dahulu. Proses pembuatan tabel kanonik dijelaskan dalam langkah-langkah sebagai berikut [6]

1. Input adalah daftar panjang kode (*codelength sequence*) dalam urutan mengecil (*descending*) $\{l_{max,}..., l_k\}$.
2. Ambil sekumpulan kode (*words*) dengan pajang yang sama dengan nilai terbesar l_{max}. Jika terdapat k_1 *words* dari kelompok ini, hasilkan sejumlah k_1 biner yang memiliki panjang l_{max}.
3. Jika panjang kode yang berikutnya adalah l_{k2}, pisahkan l_{k2} bit prfiks dari kode terakhir dari kelompok panjang kode yang sebelumnya. Tambahkan 1 sebanyak k_2 kali, dimana k_2 adalah jumlah *words* yang memiliki panjang kode l_{k2} untuk mendapatkan kode dari kelompok panjang kode tersebut.
4. Ulangi proses di atas untuk semua kelompok panjang kode.

Untuk lebih jelasnya, mari kita simak contoh berikut ini. Kita akan menggunakan sebagian data dari tabel pada contoh sebelumnya yang ditulis ulang sebagai berikut.

Simbol	**Kode Huffman Kanonik**	**Panjang Kode**
a	00000	5
b	00001	5
c	00010	5
d	00011	5
e	001	3
f	01	2
g	10	2

h	11	2

Tabel 5. Kode Huffman kanonik

Dari tabel di atas kita dapatkan daftar panjang kode (*codelength sequence*) adalah (5,5,5,5,3,2,2,2).

Algoritma untuk menghasilkan tabel kanonik benar-benar sama dengan lagkah-langkah yang sudah dijelaskan di atas.

Kita nyatakan first(l) sebagai kode pertama dalam kelompok kode dengan panjang kode l. Untuk keperluan pembuatan table kanonik, kita hanya membutuhkan first(l) untuk nilai-nilai l yang sama dengan $l_1, l_2, ..., l_{max}$ yang mana merupakan panjang kode. Tetapi, kita akan menghitung first (l) untuk semua nilai l dalam selang $l_1 \leq l \leq l_{max}$ karena, seperti yang akan kita lihat nanti, kita akan membutuhkannya pada saat proses *decoding*. Kita nyatakan juga num(l) sebagai jumlah kode dengan panjang kode l, $l_1 \leq l \leq l_{max}$. Penghitungan (komputasi) first(l) diberikan dalam potongan kode program berikut ini [6].

```
first(l_max]) := 0;
for l := (l_max -1) down to 1 do
   begin
      first(l) :=
      floor((first(l+1)+num(l+2))/2);
   end;
```

Jika kode program di atas kita terapkan pada darftar panjang kode (5,5,5,5,3,2,2,2), akan diperolah hasil sebagai berikut

first (5) = 0
first (4) = ⌈ (0 + 4) / 2 ⌉ = 2
first (3) = ⌈ (2 + 0) / 2 ⌉ = 1
first (2) = ⌈ (1 + 1) / 2 ⌉ = 1
first (1) = ⌈ (1 + 3) / 2 ⌉ = 2

dan kita peroleh tabel berikut

l	1	2	3	4	5
num(l)	0	3	1	0	4
first(l)	2	**1**	**1**	2	**0**

Tabel 6. Tabel kanonik

Hanya yang dicetak tebal dalam array first(l) yang akan digunakan. Ekspresi ⌈ (first(l+1) + num(l+2))/2 ⌉ pasti menghasilkan kode prefiks.

Sekarang akan ditunjukkan algoritma untuk menyimpan nilai first(l) yang dihasilkan ke dalam lokasi yang kontigu dalam memori utama, mulai dari address 0. Sifat inilah yang akan membuat proses *decoding* menjadi sangat efisien seperti yang akan kita lihat nanti. Kode program ini akan menghasilkan address first(l) untuk setiap nilai l. Akan dihasilkan sebuah array first_address(l).

```
begin
   first_address(l_max) ← 0;
   next_available_address ← 0 +
   num(l_max);
   for l = (l_max – 1) down to 1 do
      if num(l) ≠ 0 then
         first_address(l) ←
         next_available_address;
         next_available_address ←
         first_address(l) + num(l);
      else
         first_address(l) ← 0;
end
```

kita peroleh tabel sebagai berikut

l	1	2	3	4	5
first(l)	2	**1**	**1**	2	**0**
first_address(l)	0	**5**	**4**	0	**0**

Tabel 7. Tabel kanonik

Perhatikanlah address untuk l = 4 dan l = 1 diset menjadi 0 dan direpresentasikan sebagai dummy addresses, tetapi tidak berguna dalam proses *decoding*.

2.5. Proses *Decoding*

Sekarang, kita akan melakukan operasi *decoding* dengan string bit sebagain input. Kita definisikan v sebagai sebuah variabel penyimpan input bits (yang akan di-*decode*) sepanjang nilai biner yang direpresentasikan oleh v lebih kecil dari first(l). Catatlah bahwa kita akan tetap membutuhkan nilai first(l) walaupun tidak terdapat kode dengan panjang kode l.

Saat nilai v bertambah besar melebihi atau sama dengan nilai first(l), itu berarti loop program sedang berada di antara dua buah kelompok kode (kelompok kode berdasarkan kesamaan panjang kode), jadi, kita harus menentukan batas address kelompok kode ini untuk mengakses simbol yang disimpan pada array dalam memori. Berikut ini adalah algoritmanya.

```
while input not exhausted do
   l = 1;
   simpan input-bit dalam v;
   while v < first(l) do
      rangkaikan input-bit
      selanjutnya di belakang v;
```

```
    l = l + 1;
difference = v – first(l);
output simbol pada address
first_address(l) +
difference;
```

Catatlah bahwa untuk setiap simbol yang di-*decode*, kita hanya membutuhkan satu kali akses memori. Sedangkan untuk pohon Huffman, dibutuhkan akses memori untuk setiap bit.
Berikut ini adalah tabel kanonik dan simbol yang terdapat di memori dari contoh yang dikerjakan di atas.

Tabel kanonik

l	1	2	3	4	5
num(l)	0	3	1	0	4
first(l)	2	**1**	**1**	2	**0**
first_address(l)	0	**5**	**4**	0	**0**

Tabel 8. Tabel kanonik

Tabel address simbol di memori

Address	Simbol	Kode
0	a	00000
1	b	00001
2	c	00010
3	d	00011
4	e	001
5	f	01
6	g	10
7	h	11

Tabel 9. address simbol di memori

Untuk menunjukkan proses *decoding* kode Huffman knonik akan di-*decode* sebuah potongan string biner berikut ini:
00001000000001000000000110010001011
Mengikuti algoritma yang sudah dijelaskan sebelumnya, proses *decode* untuk simbol pertama adalah:

1. Nilai l = 1
2. Assign bit pertama, yaitu 0, pada variabel v.
3. Nilai representasi biner dari v adalah 0 yang lebih kecil daripada first(1) =2, maka rangkaikan bit berikutnya, yaitu 0, sehinggan nilai v = 00. Tambahkan nilai l dengan 1.
4. Nilai l = 2.
5. Nilai representasi biner dari v adalah 0 yang lebih kecil daripada first(2) =1, maka rangkaikan bit berikutnya, yaitu 0, sehinggan nilai v = 000. Tambahkan nilai l dengan 1.
6. Nilai l = 3.
7. Nilai representasi biner dari v adalah 0 yang lebih kecil daripada first(3) =1, maka rangkaikan bit berikutnya, yaitu 0, sehinggan nilai v = 0000. Tambahkan nilai l dengan 1.
8. Nilai l = 4.
9. Nilai representasi biner dari v adalah 0 yang lebih kecil daripada first(4) =2, maka rangkaikan bit berikutnya, yaitu 1, sehinggan nilai v = 00001. Tambahkan nilai l dengan 1.
10. Nilai l = 5.
11. Nilai representasi biner dari v adalah 1 yang lebih besar daripada first(5) = 0, maka stop condition terpenuhi dan loop berhenti..
12. Nilai difference = 1 – 0 = 1.
13. Output simbol pada address ke first_address(5) + difference = 0 + 1 = 1. Dapat dilihat pada tabel, simbol dengan address 1 adalah ’b’.
14. Nilai l diinisialisasi kembali menjadi 1 dan program siap me-*decode* simbol berikutnya.

Dengan melakukan langkah yang serupa pada simbol ketiga sampai ketujuh, kita akan memperoleh hasil
00001 00000 00010 00000 00011 001 00010 11 di-*decode* menjadi ”bacadec”.
Untuk dua bit terakhir atau sama dengan satu simbol terakhir, langkah-langkahnya akan dijabarkan sebagai berikut:

1. Nilai l = 1
2. Saat ini, program sudah sampai pada bit ke-34 yang akan kita sebut current bit. Assign current bit, yaitu 1, pada variabel v.
3. Nilai representasi biner dari v adalah 1 yang lebih kecil daripada first(1) = 2, maka rangkaikan bit berikutnya, yaitu 1, sehinggan nilai v = 11. Tambahkan nilai l dengan 1.
4. Nilai l = 2.
5. Nilai representasi biner dari v adalah 3 yang lebih besar daripada first(2) = 1, maka stop condition terpenuhi dan loop berhenti.
6. Nilai difference = 3 – 1 = 2.
7. Output simbol pada address ke first_address(2) + difference = 5 + 2 = 7. Dapat dilihat pada tabel, simbol dengan address 7 adalah ’h’.
8. Lihatlah bahwa string biner telah habis sehingga stop condition kalang loop terluar terpenuhi dan program berhenti.

Hasil akhir proses *decode* yang kita dapatkan adalah ”bacadech” yang merupakan hasil *decoding* dari string biner 00001 00000 00010 00000 00011 001 00010 11.

4. Peningkatan Efisiensi

Kode Huffman kanonik merupakan variasi dari kode Huffman yang lebih efisien dalam berbagai hal. Berikut ini akan dijelaskan letak efisiensi yang dihasilkan kode Huffman kanonik terhadap kode Huffman.

4.1. Lebih Efisien dalam Ukuran Data saat Transmisi Data

Untuk menjelaskan hal ini, mari kita misalkan:
A = 11
B = 0
C = 101
D = 100

Ada berbagai cara untuk menuliskan kode prefiks dari pohon Huffman di atas. Misalkan kita tuliskan setiap simbol, kemudian diikuti oleh jumlah bit (panjang kode) dan kode itu sendiri:
(‘A’,2,11), (‘B’,1,0), (‘C’,3,101), (‘D’,3,100)
Jika kita membuat list secara terurut berdasarkan alfabet, kita tidak perlu menuliskan simbolnya karena akan kita ketahui dengan sendirinya. List yang kita dapatkan:
(2,11), (1,0), (3,101), (3,100)
Dengan kode versi kanonik ini kita dapat mengetahui bahwa simbol sudah diurutkan berdasarkan abjad dan bahwa nilai suatu kode akan selalu lebih besar daripada kode yang mendahuluinya Ini menyebabkan bagian yang tersisa untuk ditransmisi hanyalah jumlah bit (panjang kode) untuk setiap simbol. Catatlah bahwa pohon Huffman kanonik selalu memiliki nilai yang lebih besar untuk bit yang lebih panjang dan untuk setiap simbol dengan panjang kode yang sama, selalu memiliki nilai kode yang lebih tinggi untuk simbol yang lebih tinggi dari segi urutannya.
A = 10 (code value: 2 decimal, bits: 2)
B = 0 (code value: 0 decimal, bits: 1)
C = 110 (code value: 6 decimal, bits: 3)
D = 111 (code value: 7 decimal, bits: 3)

Jadi, yang perlu kita sertakan dalam proses transmisi dta hanyalah panjang kodenya saja untuk setiap simbol.:
(2, 1, 3, 3)

Perhatikanlah bahwa hal ini sesuai dengan yang diproses oleh algoritma *encoding*, dimana yang menjadi masukan hanyalah list panjang kode.
Dengan ini, sangat memungkinkan untuk membangun kembali keseluruhan tabel (simbol dan panjang kode) hanya dengan mengetahui panjang kode yang tersusun terurut. Simbol-simbol yang tidak terdapat dalam file yang dikompresi pada umumnya diset memiliki panjang kode nol. [9]
Pada kode Huffman, proses transmisi harus menyertakan pohon Huffman sebagai alat untuk men-*decode* string biner. Hal ini jelas membuat ukuran data menjadi besar.
Dengan hanya mengirimkan list panjang kode pada saat transmisi data, proses transmisi dapat dilakukan secara lebih efisien karena ukuran data dibuat lebih kecil.

4.2. Lebih Efisien dalam Sinkronisasi terhadap Kesalahan (*Error*) [5]

Adakalanya kesalahan terjadi baik pada proses *encoding* maupun *decoding*. Misalnya terlewat satu atau beberapa bit dari string biner.
Pohon kanonik tersinkronisasi lebih baik terhadap kesalahan karena setiap sub-pohon dari pohon kanonik merupakan pohon kanonik juga.
Perkiraan jumlah bit setelah terjadi kesalahan sampai dilakukan sinkronisasi dijelaskan oleh kesamaan berikut

$$E = \frac{W}{P(S)}$$

dengan W adalah rata-rata panjang kode dan P(S) adalah peluang kejadian dimana titik sinkronisasi terletak di akhir kode yang mengandung simpul x dimana x adalah anggota himpunan simpul-simpul dalam.
Adapun penjelasan perhitungan secara lebih rinci tidak akan dijelaskan pada makalah ini.

4.3. Lebih Efisien dalam Proses *Decoding*

Ini merupakan kelebihan kode Huffman kanonik yang paling signifikan. Pada bagian ini, pross *decoding* tidak akan dijelaskan karena sudah terdapat pad bagian sebelumnya dari makalah ini.
Dari contoh pada bagian penjelasan proses *decoding*, kita dapat melihat bahwa kode Huffman kanonik memiliki proses *decoding* yang sangat efisien, hanya melakukan satu kali akses memori untuk setiap simbol yang di-*decode*, sisanya adalah proses komputasi. Di lain

pihak, kode Huffman “biasa” membutuhkan akses memori pada setiap bit-nya.
Proses *decoding* pada kode Huffman kanonik memang menuntut komputasi yang lebih banyak dari kode Huffman ”biasa”, namun ini jauh lebih efisien daripada proses *decoding* pada kode Huffman “biasa” yang melakukan traversal pohon Huffman untuk setiap bit pada string biner.
Perbedaan efisiensi kode Huffman kanonik dengan kode Huffman “biasa” akan sangat signifikan pada file dengan ukuran alfabet yang besar [6].

5. Kesimpulan

Kode Huffman kanonik memiliki keuntungan dalam hal efisiensi. Keuntungan yang paling signifikan adalah dalam hal proses *decoding*, terutama pada file dengan ukuran alfabet yang besar. Kode Huffman kanonik memerlukan akses memori yang jauh lebih sedikit daripada kode Huffman “biasa”
Keuntungan lain adalah ukuran data yang kecil. Hal ini menguntungkan terutma pada saat transmisi data. Tidak perlu mengirimkan pohon Huffman, melainkan cukup dengan daftar panjang kode tiap simbolnya.
Dalam hal sinkronisasi terhdap kesalahan, kode Huffman kanonik memiliki waktu sinkronisasi yang lebih cepat.

DAFTAR PUSTAKA

[1] Munir, Rinaldi. (2006). Diktat Kuliah IF2153 Matematika Diskrit. Program Studi Teknik Informatika, Institut Teknologi Bandung

[2] Wardoyo, Irwan. dkk. “*Kompresi Teks dengan Menggunakan Algoritma Huffman”*. Jurusan Teknik Informatika, Sekolah Tinggi Teknologi Telkom, Bandung

[3] *Anonim.* slide presentasi “*Signal Compression*“ (31/08/2006). Tidak dicantumkan nama instansi

[4] Hartawan, Ketut. dkk. *“Pemampatan Data dengan Algoritma Huffman Kanonik”*. Jurusan Teknik Informatika, Sekolah Tinggi Teknologi Telkom, Bandung

[5] Klein and Wiseman. slide presentasi *“Static Huffman”*. Tidak dicantumkan tanggal dan nama instansi

[6] *Anonim. ”Static Codes”*. Tidak dicantumkan tanggaldan nama instansi

[7] Fauzi, Ismail. slide presentasi *“Perkakasan Komunikasi Data”*. Tidak dicantumkan tanggal dan nama instansi

[8] Wikipedia (2006). http://www.en.wikipedia.org/wiki/Huffman_coding. Tanggal akses: 29/12/2006 pukul 15.00

[9] Wikipedia (2006). http://www.en.wikipedia.org/wiki/Canonical_Huffman_code. Tanggal akses: 3/1/2007 pukul 10.19